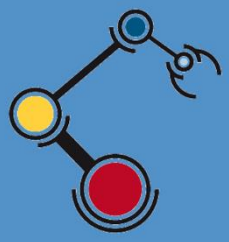
Escuela Politécnica Superior

Control cinemático de robots: Estrategias de control Articular y Cartesiano sobre un robot UR3



Máster Universitario en Automática y Robótica

Trabajo Fin de Máster

Autor:
Alejandro Díaz Reig

Tutor/es:
Gabriel Jesús García Gómez

Junio 2021

Universitat d'Alacant
Universidad de Alicante

Control cinemático de robots: Estrategias de control Articular y Cartesiano sobre un robot UR3

Autor

Alejandro Díaz Reig

Tutor

Gabriel Jesús García Gómez

Dept. Física, Ingeniería de Sistemas y Teoría de la Señal

Máster Universitario en Automática y Robótica

Escuela Politécnica Superior

Universitat d'Alacant
Universidad de Alicante

Alicante, junio 2021

Resumen

El UR3 es un robot colaborativo de 6 grados de libertad. El principal problema para crear esquemas de control cinemático para los robots comerciales es que, para ello, se debe trabajar a muy alta frecuencia. Los robots industriales muchas veces no permiten enviar nada más que posiciones articulares. Los UR3 de Universal Robots permiten además enviar velocidad articular. Sin embargo, para controlar el movimiento de las articulaciones a través de un dispositivo externo al controlador, se deben tener muchos aspectos de comunicación en cuenta. En este proyecto se desarrollarán los principales esquemas de control cinemático existentes para robots manipuladores, aplicándolos sobre el UR3 de Universal Robots.

Resum

El UR3 és un robot col·laboratiu de 6 graus de llibertat. El principal problema per a crear esquemes de control cinemàtic per als robots comercials és que, per a això, s'ha de treballar a molt alta freqüència. Els robots industrials moltes vegades no permeten enviar res més que posicions articulars. Els UR3 d'Universal Robots permeten a més enviar velocitat articular. No obstant això, per controlar el moviment de les articulacions a través d'un dispositiu extern al controlador, s'han de tenir molts aspectes de comunicació en compte. En aquest projecte es desenvoluparan els principals esquemes de control cinemàtic existents per a robots manipuladors, aplicant-los sobre el UR3 d'Universal Robots.

Abstract

The UR3 is a 6 degrees of freedom collaborative robot. The main problem in creating kinematic control schemes for commercial robots is that, to do this, you must work at a very high frequency. Industrial robots often do not allow sending anything other than joint positions. The UR3s from Universal Robots also allow to send joint speed. However, to control joint movement through a device external to the controller, many communication aspects must be taken into account. In this project the main existing kinematic control schemes for manipulator robots will be developed, applying them to the UR3 from Universal Robots.

Justificación

La motivación para realizar este trabajo ha sido la de "dar vida al robot", es decir, la estructura mecánica en conjunto con los motores tiene que ser manejada de alguna forma, de modo que diseñar los controladores articulares y Cartesianos para el UR3 permite que el robot sea capaz de seguir referencias de posición y velocidad para situarlo donde se desee.

Debido a la implementación de los controladores, el robot será capaz de desplazar cualquiera de sus 6 articulaciones a la posición que se desee dentro de su rango de trabajo.

Además, será capaz de desplazar su efector final en las tres direcciones de movimiento xyz, gracias al controlador Cartesiano, que requiere de todas las articulaciones para mover el extremo.

La robótica está en auge y como ingeniero en electrónica y automática, este proyecto me permite ir más allá de los conocimientos adquiridos en el grado, de forma que lograr implementar este tipo de controladores abre un abanico enorme de posibilidades hacia el mundo laboral.

No solo por estos factores, sino por el mero hecho de ver que una estructura mecánica motorizada se comporta acorde a lo que se quiere, es muy satisfactorio, de ahí, las ganas y el entusiasmo que se han plasmado en el presente Trabajo Final de Máster.

Agradecimientos

Quisiera agradecer a mi tutor, Gabriel, gracias a su apoyo y motivación durante estos meses de investigación el proyecto ha podido llevarse a cabo, más que un tutor ha sido un mentor, siempre transmitiéndome que nunca hay que desistir por más turbio que se vea el asunto, con trabajo y esfuerzo florecen los resultados. Por estas razones le estoy más que agradecido. Gracias.

Como no, a mis padres, por enseñarme lo que es el sacrificio, el compromiso y la lealtad. Siempre me habéis motivado a seguir mejorando y prometo seguir adelante, cueste lo que cueste. Vosotros siempre me habéis brindado todo lo necesario durante estos años para llegar a la cima de esta montaña, os estoy eternamente agradecido, seguiré volando.

A todos mis familiares y gente que me quiere, esto no solo es por mí, también por vosotros.

Gracias por enseñarme lo que es la humildad y nunca dejar de apoyarme, esto también va por ti, Anas.

Dedicatoria

Al meu iaio Alejandro, que des de la seua casa redacte aquestes paraules.

Aguarde que continues veient-me créixer i guiant-me allà on estigues.

Cita Célebre

Si tu odio se convirtiera en electricidad, se iluminaría todo el mundo.

Nikola Tesla.

Índice de Contenido

Índice de Figuras

Índice de Tablas

Índice de códigos

1. Introducción

La robótica, posee densas raíces culturales a lo largo de la historia humana. De forma que siempre se ha intentado encontrar suplentes que puedan imitar nuestro comportamiento (humano), en varias situaciones donde surge la necesidad de interactuar con el entorno.

Una de las principales metas de los humanos ha sido la de hacer cobrar vida sus diseños, ya demostrado desde la mitología griega, donde se forjaban esclavos con bronce, tras el paso de los años, Karel Čapek, en su obra *Robots Universal Rossum [1]* datada en 1920, introdujo el concepto de autómata como un artefacto mecánico capaz de sustituir o replicar labores que debían ser realizadas por humanos. En esta obra, Čapek nombró el término *robot* por primera vez refiriéndose al autómata construido por Rossum, quien acaba victorioso frente a la humanidad en esta historia de carácter fantástico.

A diferencia de los que se conoce por la robótica actual, los robots creados por Rossum no eran más que acoples mecánicos fabricados de material orgánico. No fue hasta los años 1940 cuando el escritor ruso Isaac Asimov, describió a un robot como un automatismo que debía asimilarse tanto exterior como interiormente a un humano, siendo el paradigma, que el comportamiento de dichos robots vendría dado por la programación en función del conocimiento humano donde estableció una serie de normas éticas que hoy se conocen como *Las Tres Leyes Fundamentales de la Robótica*, propuestas en su libro *I Robot [2]*.

Dichas leyes soportan lo siguiente:

1. Un robot no hará daño a un ser humano ni, por inacción, permitirá que un ser humano sufra daño.

2. Un robot debe cumplir las órdenes dadas por los seres humanos, a excepción de aquellas que entren en conflicto con la primera ley.

3. Un robot debe proteger su propia existencia en la medida en que esta protección no entre en conflicto con la primera o con la segunda ley.

De forma que estas leyes conformaran un principio organizador y unificador para las historias de ciencia ficción previas a su época donde ya se había nombrado el concepto de autómata. Desde entonces hasta la actualidad, un robot, tiene la connotación de ser un producto industrial, diseñado por ingenieros, con función de suplir tareas repetitivas.

En los últimos 10 años, esta connotación está cambiando debido a la presencia de la inteligencia artificial, que poco a poco se está abriendo el paso, para dejar de lado el concepto de que los robots solo sirven si se emplean para realizar tareas sencillas, de forma que a los robots se les pueda enseñar a aprender, no a memorizar, es decir, que ellos mismos sean capaces de tomar decisiones frente a las situaciones a las que se vean expuestos.

Por otra parte, cabe destacar el novedoso concepto de cobots, diseñados para trabajar con las personas, no solo para suplir sus necesidades. En general, estos cobots poseen sensores de fuerza que permiten detectar colisiones y detenerse en caso de haber ocurrido algún acontecimiento no contemplado en su programa, estéticamente diseñados para que, si colisionan, los daños sean los menores posibles, ya que son diseños redondeados, a diferencia de los de carácter industrial. Además, trabajan a velocidades más reducidas lo que les hace idóneos para el trato con humanos.

Figura 1: Comparativa visual robot (Fanuc M-10) frente a un cobot (UR UR3)

La mayoría de los robots industriales, simplemente permiten enviar posición articular, de forma que controlarlos externamente resulta inviable porque los mismos fabricantes proporcionan el software y hardware necesario para realizarles la programación que se requiera en función de la tarea que deseen realizar.

Del otro lado, los cobots de *Universal Robots* permiten además de enviar posición articular, posición del efector final, velocidades articulares y cartesianas leer aceleraciones, temperaturas de los motores de las articulaciones, fuerza-par, voltajes, corrientes, … lo que los hace ideales para ser controlados externamente, ya que tenemos acceso a los diferentes parámetros que se requieren para actuar sobre ellos de manera remota.

1.1. Objetivos

Dado que el UR3 de *Universal Robots* permite enviar un paquete con toda su información, del cual se puede extraer la que se requiera, se planteó la posibilidad de controlar el robot externamente, es decir, diseñar los controladores articulares y cartesianos que permitan que el robot sea capaz de seguir diferentes trayectorias, tanto para una articulación concreta como para el efector final, que implica el movimiento de todas las articulaciones.

El proyecto consta principalmente de 2 objetivos:

El primero, consiste en diseñar y simular los controladores (articular y cartesiano) en MATLAB, de forma que mediante la API remota que proporciona CoppeliaSim, se hará la comunicación entre los programas vía *socket*, luego, introduciendo un UR3 en la escena del simulador, se podrá verificar el correcto funcionamiento de los controladores, si el robot es capaz de seguir las referencias correctamente tanto en posición como en velocidad.

El segundo, tras haber realizado las simulaciones y comprobado que el UR3 se puede controlar externamente, trata de implementar de nuevo los controladores, esta vez trabajando sobre el robot real, mediante comunicación TCP/IP a través de un *socket*, con la finalidad de poder corroborar que además de poder diseñar los controladores y conseguir seguimientos correctos en simulación también se puede hacer para el robot real.

Por tanto, el objetivo del proyecto consiste en dar un paso más allá en el control de robots manipuladores, siendo capaces de obtener las lecturas del robot a tiempo real, diseñar los controladores y en función de esas lecturas, hacer el envío de velocidad correspondiente para que el robot corrija el error en posición y sea capaz de seguir referencias articulares y cartesianas, tanto en simulación como en la realidad.

1.2. Estructura del proyecto

El presente TFM, está dividido en cinco partes diferenciadas con la finalidad de separar y enfatizar de forma clara y concisa los distintos conceptos tratados.

De forma que, en el primer capítulo de introducción, se ha tratado acerca de los primeros reflejos del ser humano ante el concepto que se conoce hoy en día como robótica, diferenciando entre los robots de carácter industrial y el novedoso concepto de cobots, estableciendo las principales diferencias entre ellos, resaltando, que están explícitamente

diseñados para trabajar juntamente con personas, no como el caso de los robots industriales que buscan suplir el trabajo humano, básicamente, sustituyendo tareas repetitivas como puede ser un *pick and place*.

Seguidamente, se abordará la situación del estado del arte actual, en el ámbito de control de robots, tratando acerca del control cinemático y dinámico donde se mostrarán las principales características y diferencias entre las principales técnicas empleadas para control de robots en la actualidad, haciendo hincapié en el primero, dado que es el objeto del proyecto.

A continuación, se detallará la metodología teórica y práctica empleada tanto en simulación como con el robot real para llevar a cabo el diseño e implementación de dichos controladores, tanto a nivel de software como hardware.

Tras explicar cuáles son las técnicas a emplear para conseguir la correcta comunicación entre el robot y los controladores, se procederá a hacer un análisis de los resultados donde se podrá observar las principales similitudes y diferencias de diseñar e implementar controladores cinemáticos para el UR3 de *Universal Robots* en simulación, como con el robot real.

Terminando, una vez expuestos los resultados, se argumentarán las conclusiones tras finalizar el proyecto en su totalidad de forma que se pueda realizar una comparativa entre el control de robots externamente a su controlador propio, tanto en realidad como en simulación. Finalmente, se presenta una lista de acrónimos y abreviaturas, bibliografía y anexos de códigos empleados tanto a nivel simulado como real.

2. Estado del arte

En este capítulo, se exponen los diferentes conjuntos de técnicas que, actualmente, se emplean para el control de robots, con el objetivo de ubicar el trabajo realizado en este proyecto para poder entender correctamente su lugar en el ámbito de investigación.

Se partirá, dando unas pinceladas sobre los conceptos básicos acerca del control de robots, destacando su importancia en cualquier sistema robótico. Seguidamente, se abarcarán los conceptos de control cinemático y dinámico de robots, haciendo enfoque en la primera estrategia, ya que es la utilizada en este proyecto.

Para terminar, se comentarán los conceptos de control visual y óptimo, de forma menos detallada que las mencionadas anteriormente, debido a que no son el objeto del proyecto, pero igualmente se requiere comentarlas, ya que se pueden aplicar a muchos tipos de sistemas robóticos, no solo en el ámbito de los robots manipuladores.

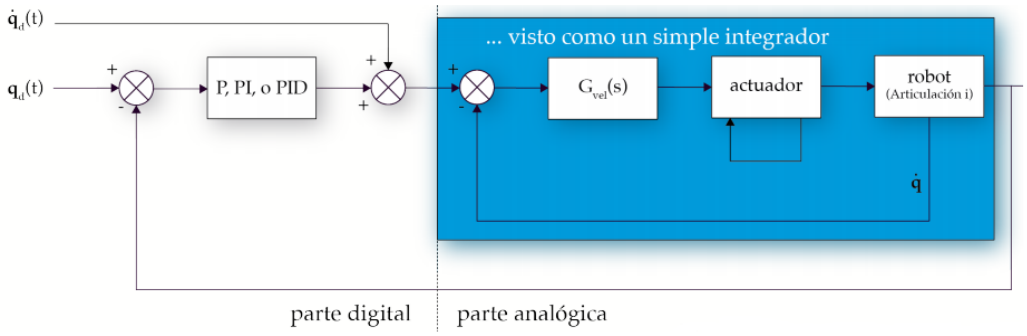


Figura 2: Bucle de control para robots industriales. Fuente: [4]

2.1. Control de Robots

En el ámbito de la robótica, para completar una determinada tarea, surge la necesidad de que el robot ejecute un movimiento específico, por tanto, la correcta ejecución de dicho movimiento depende del sistema de control que estemos aplicando, de forma que se debe de proveer al robot de las órdenes necesarias para lograr que reproduzca la trayectoria deseada.

Luego, para controlar el movimiento del robot, deben conocerse sus características tanto mecánicas como de los actuadores, de forma que objeto de este análisis, es la obtención de modelos matemáticos que describan de la mejor forma posible las relaciones existentes que caracterizan los componentes del robot. Por esta razón, se necesita de un modelo acorde a la estructura del robot, para que las relaciones matemáticas que tienen lugar desde el sistema de referencia de la base hasta el efector final sean coherentes entre sí para lograr los correctos seguimientos de trayectoria, que finalmente se traducen en buenos o malos movimientos por parte del robot.

Por tanto, en lo que al modelado se refiere, se puede distinguir principalmente entre tipos: modelado cinemático, sobre el cual está basado este proyecto sin tener en cuenta las causas que producen movimiento y modelado dinámico, que se basa en las fuerzas/pares que producen el movimiento. Ambos tipos se detallan más adelante.

2.1.1. Control Cinemático

El modelo cinemático de un robot es la descripción analítica de las relaciones entre sus posiciones articulares y las posiciones y orientaciones (cartesianas) del efector final, de forma que, conociendo las posiciones articulares, se puede calcular la posición y orientación del extremo y viceversa. Estos conceptos se conocen como cinemática directa e inversa.

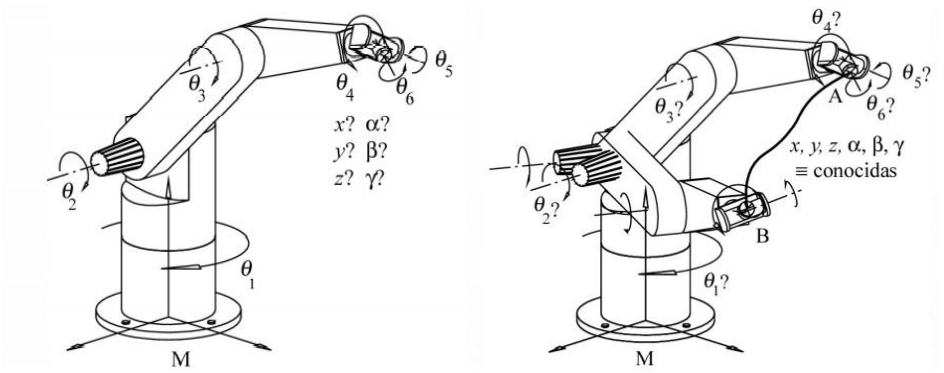


Figura 3: Cinemática directa en inversa de un robot manipulador. Fuente: [3]

En el problema cinemático directo, conocidas las posiciones articulares, se quiere obtener la posición y orientación del efector final, por tanto, solo existe una única solución ya que dado un vector de variables articulares del sistema robótico, el extremo del robot estará situado en un punto concreto del espacio.

Existen 2 métodos para resolver el problema cinemático directo:

Geométricamente, es decir, empleando las relaciones trigonométricas convenientes para obtener la posición del extremo del robot (generalmente solo se utiliza para obtener la posición no la orientación).

Mediante transformaciones homogéneas con el algoritmo *Denavit-Hartenberg* (DH) el cual asocia sistemas de referencia a cada eslabón del mecanismo robótico, calcula las transformadas homogéneas compuestas de traslaciones y rotaciones básicos para pasar del sistema asociado del eslabón *i*-1 al eslabón *i*. De esta forma, la transformación queda en función de los parámetros de la articulación *i*.

$$^{i-1}\mathbf{T}_i = \mathbf{F}(q_i) \tag{2.1}$$

Así pues, iterando las transformaciones homogéneas desde la base al extremo, se puede obtener $^{Base}T_{Extremo}$ a partir del vector q.

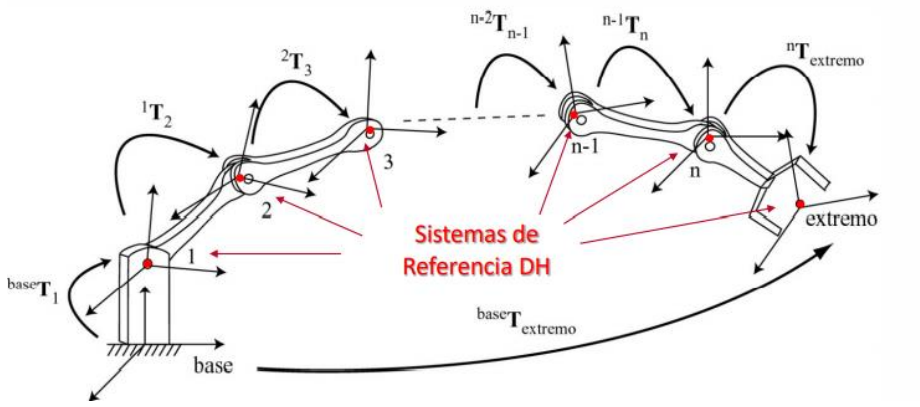


Figura 4: Sistemas de referencia algoritmo DH para un robot n articular. Fuente: [3]

Por tanto, para obtener la matriz de transformación para pasar del eslabón *i*-1 al eslabón *i*, se necesitan 4 transformaciones simples, 2 de traslación y 2 de rotación de forma que:

$$^{i-i}\mathbf{T}_i = \mathbf{Rot}(z_{i-i},\theta_i)\cdot\mathbf{Tras}(z_{i-1},d_i)\cdot\mathbf{Tras}(x_i,a_i)\cdot\mathbf{Rot}(x_i,\alpha_i) \tag{2.2}$$

Seguidamente, el problema cinemático inverso consiste en determinar las posiciones articulares del robot, siendo conocida la localización del efector final, de forma que, a partir de la posición y orientación del extremo del robot, se pueden hallar las distintas combinaciones de posiciones articulares que logren situar el extremo en dicha localización. A diferencia del problema cinemático directo, en este caso, puede que no exista solución, ya que el punto a alcanzar queda fuera del espacio de trabajo del robot o que existan múltiples soluciones, es decir, que varias configuraciones articulares permitan situar el efector final en dicha localización.

Existen 3 métodos para resolver el problema cinemático inverso:

Algebraicamente, consiste en obtener un sistema de n ecuaciones en función de la ubicación del extremo del robot, además, se puede partir de la solución proporcionada por el problema cinemático directo mediante el algoritmo DH, despejando las variables articulares de la matriz de transformación final.

Geométricamente, se busca descomponer la cadena cinemática del robot en varios planos geométricos, resolviendo por trigonometría el problema asociado a cada plano.

Por último, la solución de Pieper, la cual consiste en separar las articulaciones de la muñeca del resto, resolviendo ambos conjuntos por separado con el objetivo de simplificar los cálculos.

Por otro lado, la relación analítica entre las velocidades articulares y del extremo se conoce como cinemática diferencial, ya que el sistema de control del robot debe establecer qué velocidades debe aplicar a cada articulación para conseguir que el extremo del robot desarrolle una trayectoria temporal concreta.

De forma que estudia las relaciones entre el espacio articular y Cartesiano considerando velocidades además de posiciones.

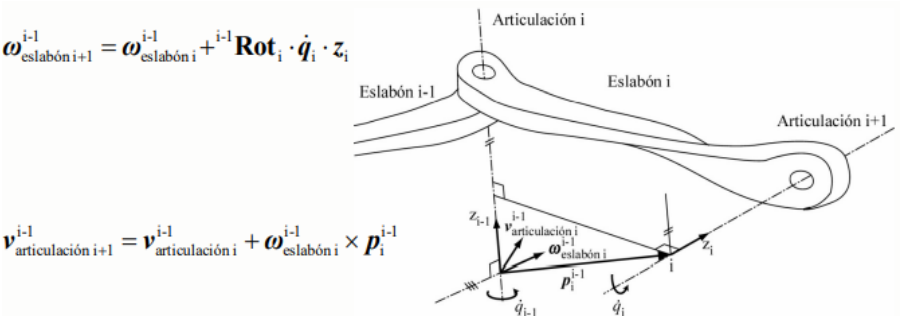


Figura 5: Esquema con las relaciones de la cinemática diferencial. Fuente: [3]

Por tanto, la matriz Jacobiana relaciona las velocidades de las coordenadas articulares con la posición y orientación del extremo del robot.

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\alpha} \\ \dot{\beta} \\ \dot{\gamma} \end{bmatrix} = \boldsymbol{J} \begin{bmatrix} \dot{q}_1 \\ \vdots \\ \dot{q}_n \end{bmatrix} = \begin{bmatrix} \frac{\partial f_x}{\partial q_1} & \cdots & \frac{\partial f_x}{\partial q_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_\gamma}{\partial q_1} & \cdots & \frac{\partial f_\gamma}{\partial q_n} \end{bmatrix} \begin{bmatrix} \dot{q}_1 \\ \vdots \\ \dot{q}_n \end{bmatrix} \tag{2.3}$$

Cabe la posibilidad de encontrarse con configuraciones singulares, esto ocurre cuando el determinante de la matriz Jacobiana es nulo, es decir, cero. Si esto ocurre, no existe Jacobiana inversa ya que no podemos invertir una matriz cuyo determinante sea nulo. Estas singularidades ocurren generalmente cuando se alinean 2 o más ejes de las articulaciones del robot.

Por tanto, para poder calcular la Jacobiana inversa, que nos permitirá hacer los envíos de velocidades articulares para producir los movimientos del extremo, habrá que tener en cuenta que en la trayectoria que queremos que describa el robot no puede haber ninguna configuración singular, ya que no podrá resolverse y el movimiento no podrá completarse.

2.1.1.1. Control Cinemático Articular

Como se ha comentado anteriormente, para planificar en el espacio articular, es decir, ejecutar una trayectoria para una única articulación del robot, no se necesita conocer la posición del extremo, pero sí la ubicación de la articulación que se desea controlar, de forma que, realimentando posición y prealimentando velocidad se puede lograr compensar el error. De esta forma, evitamos calcular la Jacobiana de forma online (a cada iteración) lo que hará que el programa sea mucho más rápido y se eviten problemas de singularidades.

Luego, generando una trayectoria, se podrá hacer el cálculo de la posición deseada menos la posición real, que, al multiplicarlo por la ganancia, aporta el término que se envía conjuntamente a la lectura, así, el robot a cada iteración es capaz de corregir el error e ir avanzando en la trayectoria hasta que se haya trazado por completo.

El esquema de un controlador cinemático articular es el siguiente:

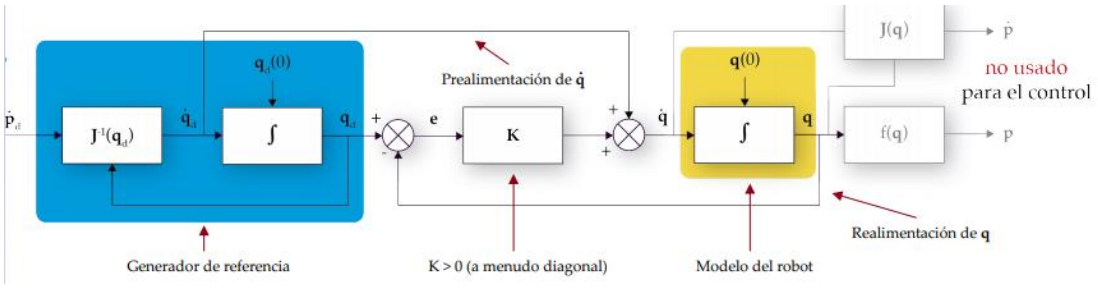


Figura 6: Esquema de un controlador cinemático articular. Fuente: [4]

Observando el esquema, se puede ver que el error es la resta entre q_d (posición deseada) menos q (posición real), que al multiplicarse por la ganancia K es el término que se suma a la prealimentación de velocidad y conforman el controlador cinemático articular que permite el seguimiento de trayectorias.

Por tanto, el cálculo de la evolución del error conduce a que, si se modela el robot como un integrador, el error evolucionaría exponencialmente en función de la ganancia K como se muestra en la expresión (2.4):

$$\mathbf{e} = \mathbf{q}_d - \mathbf{q} \Rightarrow \dot{\mathbf{e}} = \dot{\mathbf{q}}_d - \dot{\mathbf{q}} = \dot{\mathbf{q}}_d - \left(\dot{\mathbf{q}}_d + \mathbf{K}\left(\mathbf{q}_d - \mathbf{q}\right)\right) = -\mathbf{K}\mathbf{e} \tag{2.4}$$

De forma que en el instante cero el error debería ser nulo ya que el robot se encuentra inicialmente en la posición inicial de la trayectoria deseada pero para poder llevar a cabo este proceso en la realidad, existen varios problemas de comunicaciones que hacen que la trayectoria deseada deba calcularse en función del tiempo de lectura de los datos, lo que implica que la posición deseada en la primera iteración ya tenga valor, aspecto que se tratará más a fondo en el apartado de metodología.

2.1.1.2. Control Cinemático Cartesiano

Por otro lado, para llevar a cabo trayectorias en el espacio Cartesiano, debemos conocer la posición y orientación del extremo del robot, que serán los puntos de partida para la generación de la trayectoria en el efector final para los ejes *xyz* permitiendo así, el movimiento en las tres direcciones, lo que engloba el movimiento de todas las articulaciones del robot, por tanto, deberá conocerse tanto la ubicación y velocidad del efector final ya que vamos a actuar sobre él, como las posiciones articulares, ya que todas las articulaciones se verán afectadas para permitir el seguimiento de una trayectoria por el efector final.

A diferencia del controlador anterior, se requiere el cálculo de la matriz Jacobiana de forma online ya que debemos ser capaces de convertir posiciones articulares, en velocidades cartesianas para que el extremo pueda seguir la trayectoria requerida, lo que supone problemas de tiempo real ya que se incrementa el número de lecturas y operaciones en comparación al controlador anterior.

El esquema de un controlador cinemático Cartesiano es el siguiente:

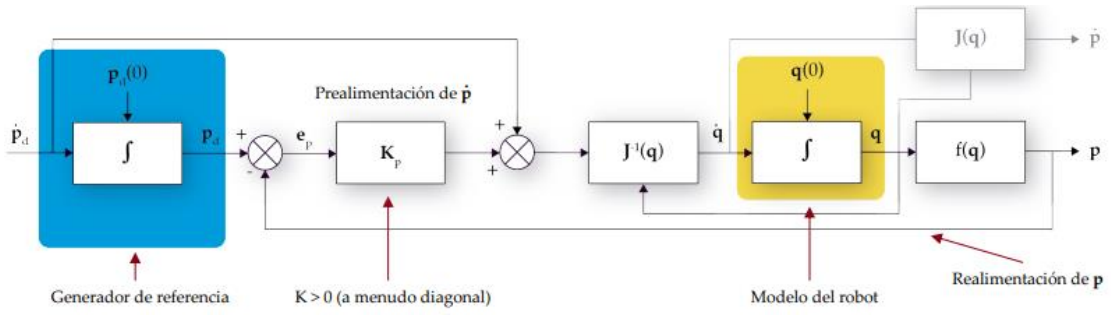


Figura 7: Esquema de un controlador cinemático Cartesiano. Fuente: [4]

Por tanto, como se observa en la Figura 7, la desviación en posición (error) se puede tratar como la velocidad deseada más el término que compensa el error, $K*(q_d - q)$ como se muestra en la expresión (2.5).

$$\mathbf{e}_p = \mathbf{p}_d - \mathbf{p} \Rightarrow \dot{\mathbf{e}}_p = \dot{\mathbf{p}}_d - \dot{\mathbf{p}} = \dot{\mathbf{p}}_d - \mathbf{J}(\mathbf{q})\mathbf{J}^{-1}(\mathbf{q})\left(\dot{\mathbf{p}}_d + \mathbf{K}_p(\mathbf{p}_d - \mathbf{p})\right) = -\mathbf{K}_p\mathbf{e}_p \quad (2.5)$$

Luego, el término $\left(\dot{\mathbf{p}}_d + \mathbf{K}_p(\mathbf{p}_d - \mathbf{p})\right)$ aplicado a cada 1 de los 3 ejes, se incluye en un vector que, al multiplicarlo por la inversa de la matriz Jacobiana, traduce las velocidades en *xyz* que se deberán enviar al robot para el correcto seguimiento de la referencia.

Igual que ocurría con el controlador cinemático articular, existen problemas de tiempo real, incluso peores que en caso anterior, ya que debemos tomar más lecturas, hacer operaciones más complejas y teniendo en cuenta que durante la trayectoria, aparezca un punto singular que el robot no pueda alcanzar y detenga su movimiento. Estos detalles se estudiarán más a fondo en el apartado de metodología.

2.1.2. Control Dinámico

El concepto de dinámica engloba la relación entre las fuerzas que actúan sobre un cuerpo y el movimiento que se origina, de forma que el modelo dinámico de un robot establece relaciones entre posiciones, velocidades y aceleraciones del robot con las fuerzas y pares aplicados a los motores de las articulaciones, lo que supone la inclusión de parámetros dimensionales y dinámicos de los eslabones, como pueden ser la longitud, las masas o los momentos de inercia.

Por esta razón, la complejidad aumenta con los GDL y la influencia de la aceleración de Coriolis de forma que estos modelos se resuelven mediante métodos numéricos donde frecuentemente se utilizan simplificaciones.

Existen 2 algoritmos para el cálculo del modelo dinámico:

El algoritmo de *Lagrange*, lo aproxima mediante consideraciones energéticas, con ecuaciones bien estructuradas, empleando la representación DH, pero tiene poca eficiencia computacional por lo que se emplea para robots de pocos GDL.

$$L = E_c - E_p$$
$$\tau_i = \frac{d}{dt}\frac{\partial L}{\partial \dot{q}_i} - \frac{\partial L}{\partial q_i} \tag{2.6}$$

Por otro lado, se puede plantear como un equilibrio de fuerzas de Newton, basado en operaciones vectoriales que emplea un algoritmo recursivo con ecuaciones poco estructuradas lo que aporta una mayor eficiencia computacional, razón por la cual se emplea para los robots con muchos GDL.

$$\sum \vec{F}(t) = m \cdot \frac{d\vec{v}(t)}{dt}$$
$$\vec{T}(t) = I \cdot \frac{d\vec{\omega}(t)}{dt} + \vec{\omega}(t) \times \left(I\vec{\omega}(t)\right) \tag{2.7}$$

Luego, el modelo dinámico de un robot, básicamente se compone de una matriz de inercias M multiplicando al vector de aceleraciones, más una matriz (n×1) de fuerzas de Coriolis, C, multiplicando al vector de velocidades más una matriz (n×1) de gravedad, G, que depende de q, de forma que τ, el vector de fuerzas o pares que se aplica a cada articulación quedaría de la siguiente forma:

$$\boldsymbol{\tau} = \mathbf{M}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \mathbf{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})\dot{\boldsymbol{q}} + \mathbf{G}(\boldsymbol{q}) \tag{2.8}$$

Además, habrá que tener en cuenta que el sistema robótico se considera un mecanismo de eslabones rígidos, en el que puede haber perturbaciones o problemas por rozamiento.

3. Metodología

En el presente capítulo, se detallarán tanto los fundamentos teóricos como prácticos en los que están basados los controladores implementados en el presente proyecto, con el objetivo de justificar su utilización y validez a nivel práctico.

Luego, en primer lugar, se explicará quienes son *Universal Robots [5]* y cuáles son los modelos de cobots que ofrecen, analizando las principales similitudes y diferencias entre las distintas series de la marca.

Seguidamente, se expondrán las herramientas necesarias para llevar a cabo mediante *MATLAB [9]* y *CoppeliaSim [8]* las simulaciones de los controladores cinemáticos articular y Cartesiano con el objetivo de verificar que el robot se puede controlar externamente siendo capaz de seguir las trayectorias que se establecen.

Tras analizar la base metódica de las simulaciones, se explicarán los principios básicos de comunicación vía TCP/IP mediante un *socket*, protocolo de comunicaciones que permite conectarse externamente al controlador del UR, mediante el cual se puede recibir la información del controlador y enviarle órdenes.

Se tratarán también los problemas de comunicaciones entre *MATLAB* y el controlador del UR3, motivo por el cual, el proyecto tuvo que redirigirse en la parte del diseño de los controladores para el robot real, ya que era inviable diseñar el controlador desde *MATLAB* a causa de los retrasos introducidos por las comunicaciones. Este tema se trata en detalle a continuación.

Tras solucionar los problemas de comunicaciones que no permitían el correcto seguimiento de la referencia en el robot real, se explicará la metodología desarrollada para llevar a cabo la implementación de los controladores articular y Cartesiano en el robot real.

3.1. Universal Robots

Universal Robots trabaja desde 2005 para marcar la diferencia en la vida de sus clientes de la manera que más les importa. Más allá de la simple automatización, *Universal Robots* cambia la forma de trabajar y vivir de las personas en todo el mundo fortaleciendo sus ideas y sueños mediante su gama de cobots.

Dentro de la gama de productos de UR, existen 2 series de cobots: La serie *e* y la serie *CB*, donde las principales diferencias están en la frecuencia a la que el controlador del robot es capaz de enviar y recibir órdenes, como se muestra en la siguiente tabla:

e-Series							
	Primary		Secondary		Real-time		Real-time Data Exchange (RTDE)
Port no.	30001	30011	30002	30012	30003	30013	30004
Frequency [Hz]	10	10	10	10	500	500	500
Receive	URScript commands	-	URScript commands	-	URScript commands	-	Various data
Transmit	See attachment from the bottom		See attachment from the bottom		See attachment from the bottom		See RTDE Guide

CB-Series							
	Primary		Secondary		Real-time		Real-time Data Exchange (RTDE)
Port no.	30001	30011	30002	30012	30003	30013	30004
Frequency [Hz]	10	10	10	10	125	125	125
Receive	URScript commands	-	URScript commands	-	URScript commands	-	Various data
Transmit	See attachment from the bottom		See attachment from the bottom		See attachment from the bottom		See RTDE Guide

Tabla 1: Comparativa parámetros control remoto en los cobots UR. Fuente: [6]

Como se puede observar en la tabla 1, en función de la serie del cobot, la frecuencia a la que trabajan los puertos de comunicaciones es distinta, prestando especial atención al puerto 30003 que es el destinado al intercambio de información en tiempo real. El robot del cual se dispone en el laboratorio es de la serie *CB*, por tanto, el envío como recepción de los datos por parte del controlador del UR3 se hará a 125 Hz, lo que corresponde a 8 milisegundos. A diferencia de las simulaciones, cuando se está trabajando con el robot real, no solo se le puede pedir al UR3 su posición y velocidad articular, por ejemplo, sino que el controlador envía un paquete con todos los datos disponibles que describen el estado en el que se encuentra el robot.

Además, no solo influye la versión del cobot que se disponga sino también la versión de controlador que tenga instalada, ya que el tamaño del paquete que envía por el puerto 30003, varía en función de la versión de *Polyscope* instalada en el *Flex Pendant*, luego UR, proporciona una hoja de datos donde en función de la versión controladora instalada en el robot, advierte de las diferencias de tamaño en el paquete de datos lo cual tiene gran importancia a la hora de realizar la lectura, como se mostrará en el apartado de problemas en las comunicaciones.

Figura 8: Diferentes modelos en función del tamaño que ofrece UR. Fuente: [7]

3.2. MATLAB y CoppeliaSim (V-Rep)

MATLAB es una plataforma de programación y cálculo numérico desarrollada por *Mathworks* empleada para analizar datos, desarrollar algoritmos y crear modelos, como es el caso, ya que es el entorno donde se implementarán los controladores cinemáticos articular y Cartesiano.

Este entorno ha sido elegido por diversas razones:

Ofrece gran versatilidad cuando se necesita trabajar con matrices, dado el potencial que ofrece la *Robotics Toolbox for MATLAB* de Peter Corke [10], un paquete especialmente diseñado para trabajar en aspectos de robótica, dicha *toolbox* ofrece un archivo llamado *UR3.mat* que incluye el modelo del robot con el que se está trabajando, lo que permite hacer cálculos de cinemática directa e inversa o Jacobianas, válidos para el control que se requiere. Además, un interpolador quíntico, *tpoly [11]*, mediante el cual se pueden generar las trayectorias que se han comentado anteriormente, lo que proporciona una gran ventaja a la hora de generar trayectorias con diferentes periodos de muestreo con duraciones distintas, además, aparte de la posición inicial y final de la trayectoria con un vector de tiempos, se pueden añadir los parámetros de velocidad inicial y final, que, por defecto, están establecidos a cero. La ventaja de este generador de trayectorias es que no solo podemos hacer que la función devuelva la trayectoria preestablecida en un periodo de muestro fijo, sino que se pueden devolver los coeficientes del interpolador de forma que la trayectoria deseada se calcule en función del tiempo de lectura, este concepto se explica más a fondo en el apartado de generación de trayectoria para el robot real.

Figura 9: Logo *MATLAB*, software desarrollado por *Mathworks.* Fuente: [9]

CoppeliaSim (anteriormente conocido como *V-REP* (Virtual Robot Experimentation Platform)) es un simulador de robótica con una extensa capacidad, funcionalidades y APIs (conjunto de funciones y métodos para ser utilizado por otro software).

Figura 10: Logo *CoppeliaSim* (*V-Rep*). Fuente: [8]

El simulador de robótica *CoppeliaSim* dispone de una interfaz o entorno de desarrollo (IDE) en la que cada modelo u objeto está basado en una arquitectura de control distribuido, de modo que cada objeto se puede controlar individualmente por un script, un plug-in, un nodo de ROS, una aplicación remota a través de su API, o una solución externa. Estos controladores se pueden desarrollar en diversos lenguajes, como *LUA* (lenguaje propio de CoppeliaSim), *C, C++, Java, Python o MATLAB*.

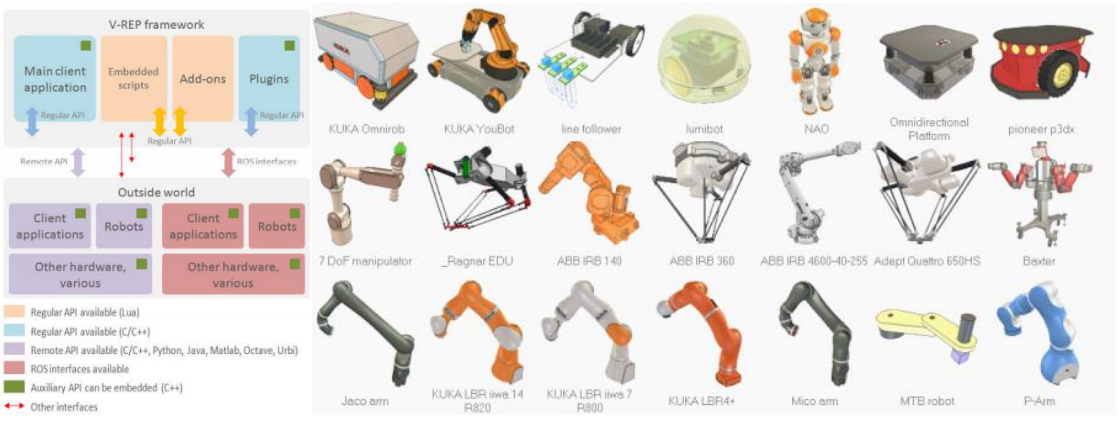


Figura 11: *Framework* y robots disponibles en *CoppeliaSim.* Fuente: [8]

Así, esto nos permite hacer la conexión vía socket entre MATLAB y CoppeliaSim haciendo uso de la API remota que proporciona el simulador. Una vez introducido el modelo de UR3 en la escena, se obtiene lo siguiente:

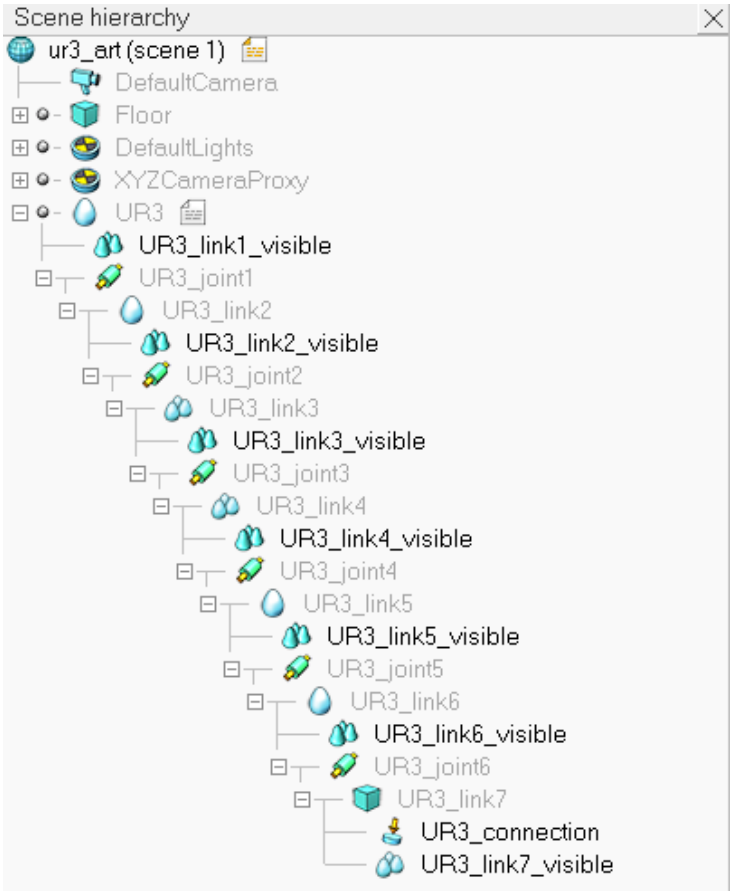


Figura 12: Jerarquía de escena de *CoppeliaSim* con el robot UR3

Tras cargar el modelo del robot, aparece su árbol jerárquico, es decir, las relaciones que tienen las articulaciones entre ellas que vienen dadas por el modelo URDF que el propio simulador permite importar.

En consecuencia, se puede visualizar al robot y acceder a todos los parámetros de las distintas articulaciones, como se muestra en la Figura 13:

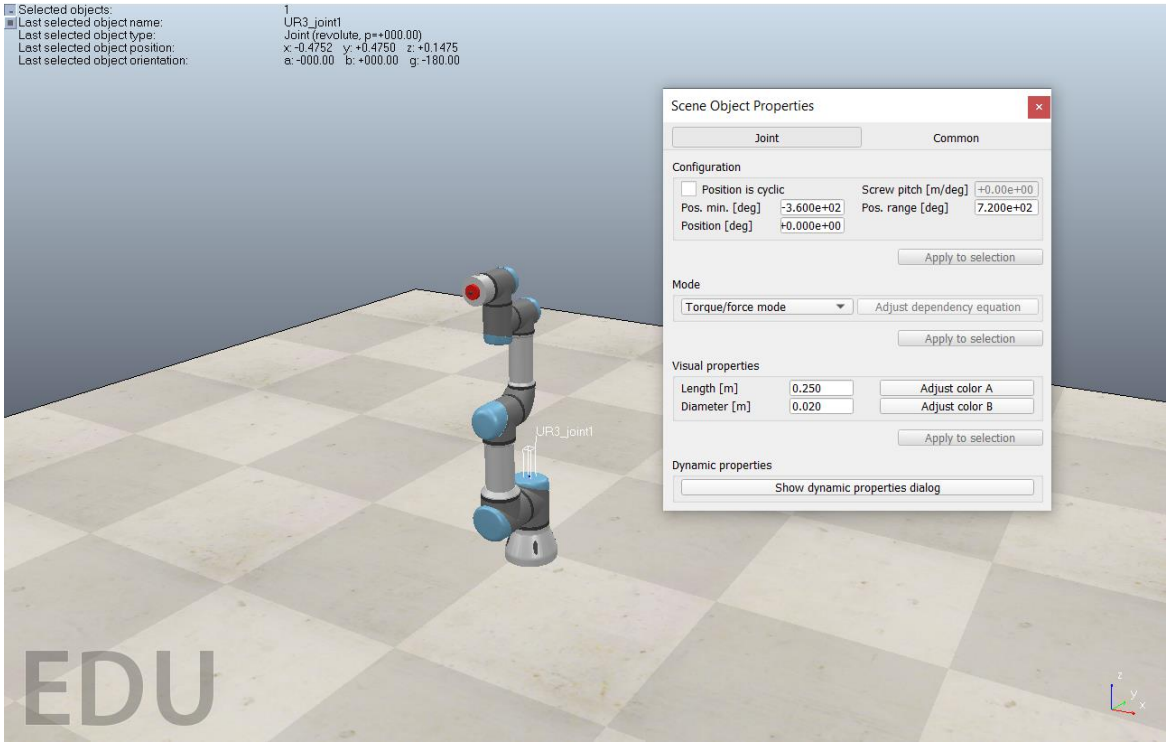


Figura 13: Escena de *CoppeliaSim* con el UR3 y las propiedades del eje de la base

Para que podamos controlar el robot externamente, es decir, desde *MATLAB* y no un script del propio simulador, tendremos que desactivar el bucle de control de la articulación de forma que tengan efecto los comandos que proporcione el controlador mediante el *socket*, como se muestra en la Figura 14. Cabe destacar que hay que mantener activo el motor de la articulación sino el simulador no será capaz de mover la articulación, ya que es como si no hubiese motor:

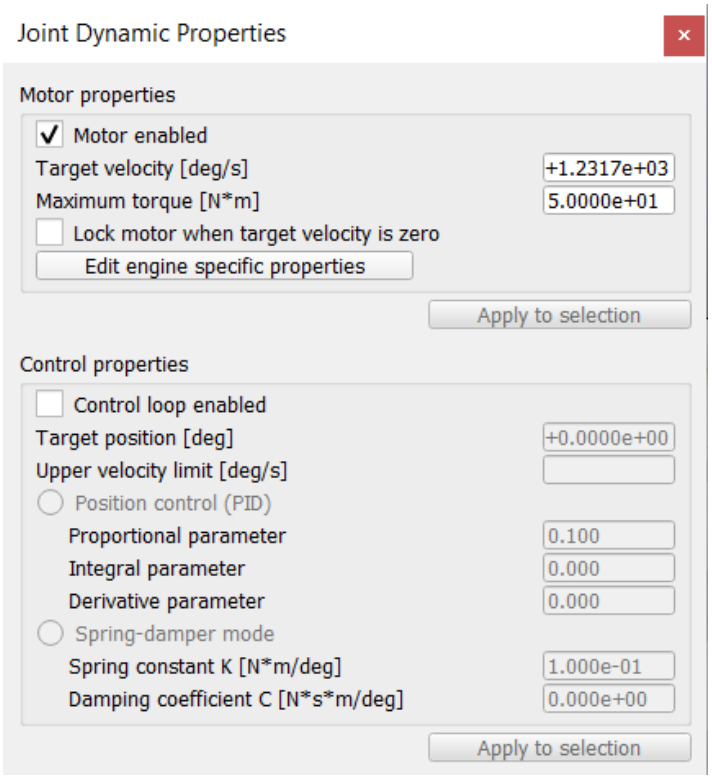


Figura 14: Propiedades dinámicas de base del UR3 en *CoppeliaSim*

Esta acción es necesario efectuarla en la articulación que se desee controlar, por tanto, como a parte del controlador cinemático articular se va a realizar un controlador cinemático Cartesiano, y todas las articulaciones del robot van a estar implicadas en mayor o menor forma en los movimientos del efector final, es conveniente realizar este procedimiento en las 6 articulaciones que tiene el UR3.

Cabe destacar también que la articulación debe estar en el modo torque/force donde la articulación se simula a través del dynamics module como se observa en la Figura 13. En este caso, se puede controlar en fuerza/par, en velocidad o en posición.

Para configurar correctamente la simulación, deberemos establecer los pasos del motor de física del simulador al mismo valor que el periodo de muestreo del generador de trayectorias del controlador. De este modo, mediante una señal de disparo a través del *socket*, el controlador desde *MATLAB* estará sincronizado con la escena y los movimientos del UR3 en *CoppeliaSim*, como se observa en las Figuras 15 y 16:

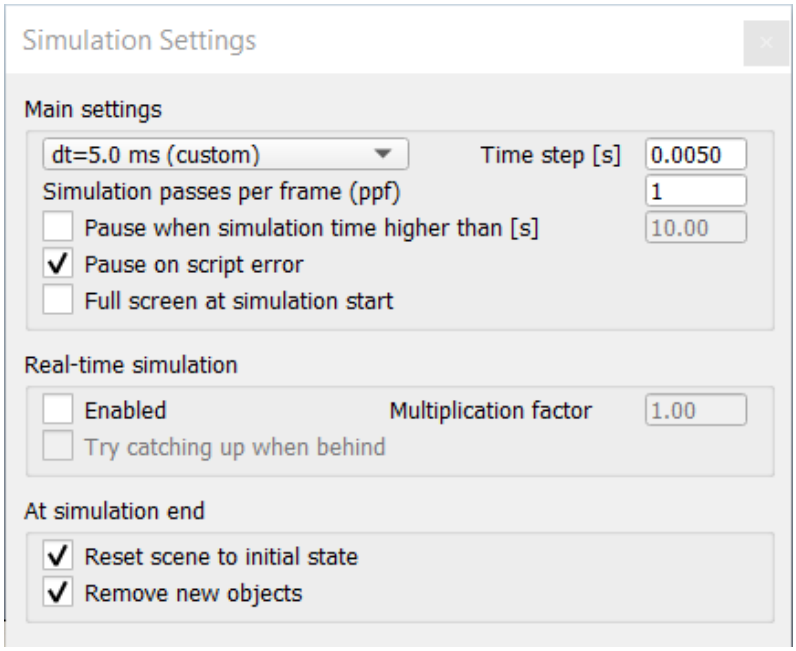


Figura 15: Configuración pasos motor de física de *CoppeliaSim*

Figura 16: Barra de Herramientas del simulador de *CoppeliaSim*

Como se puede observar en la Figura 16, el motor de física elegido es el *Bullet 2.78*, motor de código abierto que soporta detección de colisiones y simulación dinámica, con precisión en modo *balanced*, parámetro que conviene dejarlo por defecto para evitar problemas de comunicaciones una vez establecidos los pasos del motor de física.

3.3. Interpolador para la generación de la trayectoria simulada

Dado que el robot tiene la necesidad de seguir una trayectoria, surge la necesidad de generarla correctamente, para el caso del controlador cinemático articular, solamente generando un perfil de velocidad y otro de posición, ya se tendría la trayectoria deseada, pero para el caso del controlador cinemático Cartesiano, cuando se planifica una trayectoria para los 3 ángulos de *Euler*, el movimiento global resultante no se puede visualizar de manera intuitiva sin una simulación en este caso, luego, la estrategia para la generación de la trayectoria consiste en dividirla en 3 intervalos, de forma que en el primer intervalo se produzca un desplazamiento en el eje *z* (vertical), seguidamente de un desplazamiento en *y* (izquierda/derecha) y por último en el tercer intervalo, un desplazamiento en *x* (delante/atrás), de forma que visualmente se puedan apreciar los desplazamientos en línea recta del extremo que al plotear los resultados, permita una representación gráfica de los movimientos del extremo con el objetivo de poder visualizarlos correctamente.

Cabe destacar que al estar trabajando en un entorno de simulación y haber ajustado el parámetro de periodo de muestreo en *MATLAB* al mismo valor que los pasos del motor de física de *CoppeliaSim*, no existirán problemas de comunicaciones ni de tiempo real, ya que cada envío de información está determinado por una señal de disparo, que evita la desincronización entre el programa que ejecuta el controlador y el simulador donde se pueden apreciar los movimientos que se están produciendo.

Luego, para generar la trayectoria en *MATLAB*, se hará uso de la *toolbox* de robótica proporcionada por Peter Corke [10], donde incluye un interpolador quintico, ideal para conseguir los perfiles de posición, velocidad y aceleración deseados.

Así, una trayectoria válida para su seguimiento en el control cinemático articular sería la siguiente, mostrada en la Figura 9 que se genera llamado a la función tpoly, con la posición inicial de la articulación, es decir 0, la posición final que para un giro de 90 grados corresponde a 1.57 radianes y el vector de tiempos con pasos de 0.005 igual que el motor de física del simulador entre 0 y 2 segundos, de forma que se obtiene:

```
tpoly(q0,qf,t);
```

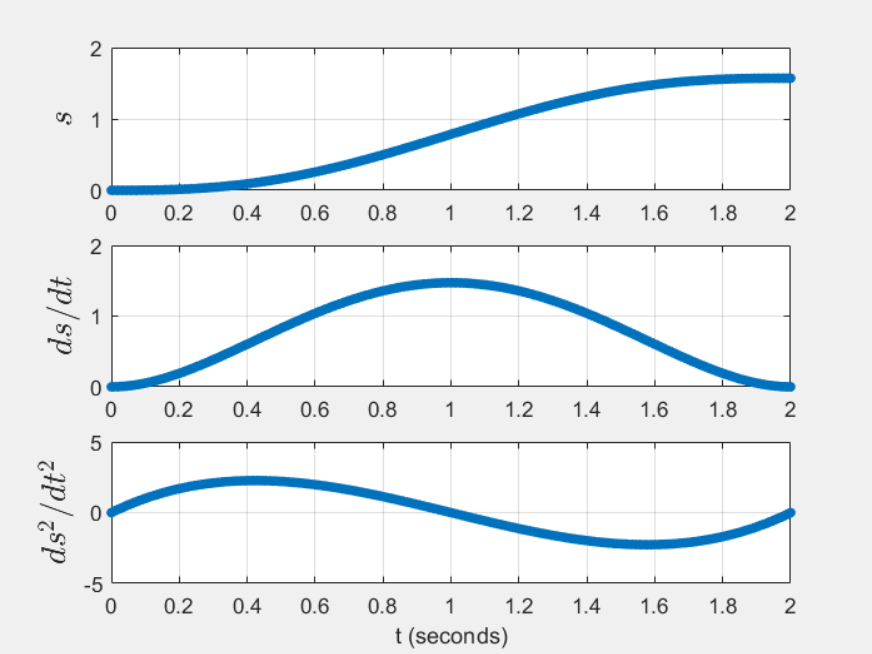


Figura 17: Trayectoria a seguir para el controlador cinemático articular

Podemos observar que la posición inicial corresponde con 0 y la final 1.57 lo que describe el giro de 90 grados correctamente, con un perfil de velocidad creciente y decreciente, dados por el perfil de aceleración.

La necesidad de emplear un interpolador quintico viene dada porque el perfil en aceleración no puede ser nulo ni constante, ya que no podría realizarse el movimiento, o no comenzaría el movimiento, o estaría continuamente acelerando, por esta razón, se ha hecho uso del interpolador de grado 5 que permite generar perfiles adecuados que se traduzcan a correctos movimientos del robot una vez implementado el controlador.

Por otro lado, como se ha comentado anteriormente, la generación de la trayectoria para el controlador cinemático Cartesiano debe ser dividida en 3 intervalos donde en cada uno de ellos se produzca el desplazamiento en el eje correspondiente, luego la forma de conseguirlo es:

```
vectorTiempo = 0:0.005:0.66;
[s1X,s1dX]=tpoly(pos(1),pos(1),vectorTiempo,0,0);
[s1Y,s1dY]=tpoly(pos(2),pos(2),vectorTiempo,0,0);
[s1Z,s1dZ]=tpoly(pos(3),pos(3)+0.1,vectorTiempo,0,0);

[s2X,s2dX]=tpoly(pos(1),pos(1),vectorTiempo,0,0);
[s2Y,s2dY]=tpoly(pos(2),pos(2)+0.1,vectorTiempo,0,0);
[s2Z,s2dZ]=tpoly(pos(3)+0.1,pos(3)+0.1,vectorTiempo,0,0);

[s3X,s3dX]=tpoly(pos(1),pos(1)+0.1,vectorTiempo,0,0);
[s3Y,s3dY]=tpoly(pos(2)+0.1,pos(2)+0.1,vectorTiempo,0,0);
[s3Z,s3dZ]=tpoly(pos(3)+0.1,pos(3)+0.1,vectorTiempo,0,0);


sX = [s1X;s2X;s3X];
sY = [s1Y;s2Y;s3Y];
sZ = [s1Z;s2Z;s3Z];
sdX = [s1dX;s2dX;s3dX];
sdY = [s1dY;s2dY;s3dY];
sdZ = [s1dZ;s2dZ;s3dZ];
```

Por tanto, pos, es un vector 1x3 con las posiciones del extremo del robot, a diferencia del controlador cinemático articular, como estamos trabajando sobre el extremo del robot, la posición inicial para el desplazamiento de cada trayectoria parte de la posición que se encuentre el efector final en ese momento, luego el resultado se muestra en la figura 10, donde se observa que en el primer intervalo se producirá un desplazamiento en z, en el segundo intervalo un desplazamiento en el eje y finalizando con el desplazamiento en el eje x.

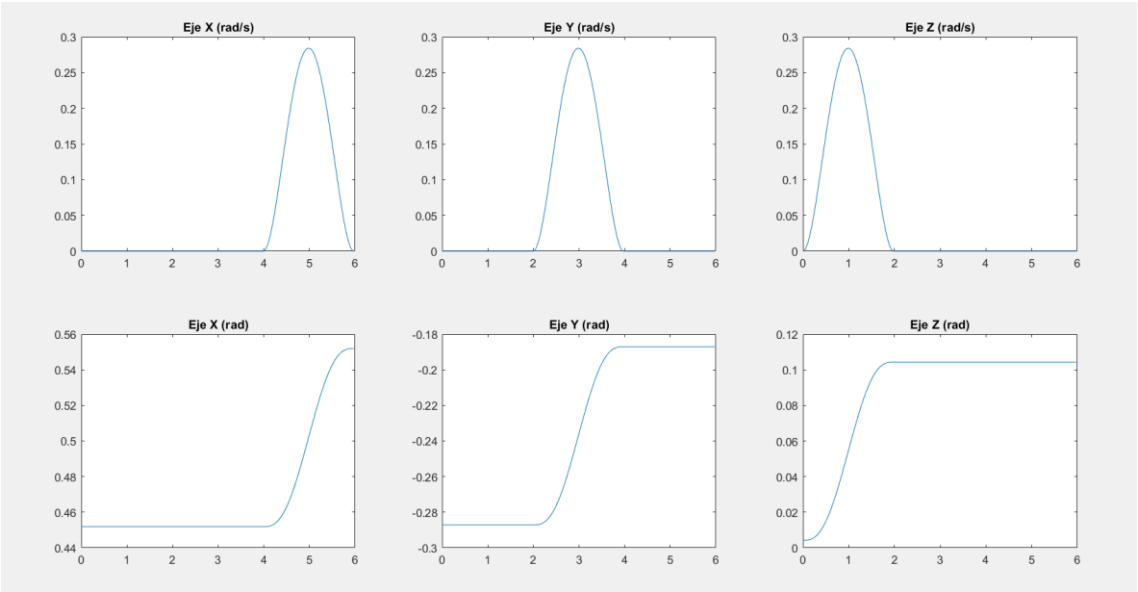


Figura 18: Trayectorias a seguir por el controlador cinemático Cartesiano

Luego, como se puede corroborar en la figura 10, los perfiles en velocidad comienzan en 0 y los perfiles en posición a partir de la posición del efector final para el robot en estado de reposo que corresponden a *0.45192* en z, *-0.28712* en y, *0.0024* en x.

3.4. Control Cinemático Articular Simulado en el UR3

Para llevar a cabo la simulación del controlador cinemático articular, se debe haber introducido un robot UR3 en la escena de *CoppeliaSim* como se muestra en la Figura 13, de forma que al haber realizado los ajustes comentados en el capítulo 3.2, por parte del simulador, solo quedaría iniciar la API remota mediante el comando que se muestra en el *Sandbox script* de la Figura 19:

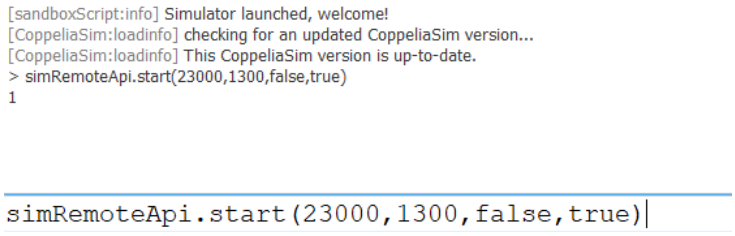


Figura 19: Inicialización de la API en el simulador

De forma que devuelve un 1 cuando el simulador está preparado para ser controlado externamente, en este caso, por el puerto 23000, con un tamaño máximo de paquete de 1300 bytes, sin *debug* y con la señal de disparo habilitada para evitar desincronizaciones.

Seguidamente, desde MATLAB hay que iniciar el servidor remoto de la API, por el puerto 23000 igual que en el simulador y con la dirección IP y demás parámetros por defecto como se muestra en la Figura 20, por tanto, si se establece la conexión (la función devuelve un 1), se tendrá la comunicación realizada entre los 2 programas:

```
disp('Inicio del programa');
vrep=remApi('remoteApi');
vrep.simxFinish(-1); % Por si hubiera una conexión anterior abierta
clientID=vrep.simxStart('127.0.0.1',23000,true,true,5000,0.01);

if (clientID>-1)
    disp('Conectado al servidor remoto de la API');
```

Figura 20: Inicialización de la API remota de *CoppeliaSim* desde *Matlab*

Tras establecer la conexión, se debe obtener el handle (nombre de referencia de la articulación en la escena) y la posición articular, de forma que MATLAB, sepa la articulación que se desea controlar y la posición de dicha articulación con el fin de generar la trayectoria como se ha comentado en el capítulo 3.3. Luego, se debe establecer la ganancia del controlador, K, y realizar las inicializaciones de los vectores, para que el programa tenga en cuenta que son variables declaradas, como se observa en la Figura 21:

```
% Seleccionamos la primera articulación del robot UR
[~,ur3]=vrep.simxGetObjectHandle(clientID,'UR3_joint1',vrep.simx_opmode_oneshot_wait);

% Obtenemos posición articular
[~,q0] =vrep.simxGetJointPosition(clientID, ur3, vrep.simx_opmode_oneshot_wait);

% Inicialización de la ganancia K

Kp = 20;

% Cálculo de la trayectoria y su velocidad
qf = q0+deg2rad(89);
t_muestreo = 0.005;
t = 0:t_muestreo:2;

[Q_deseada,Qd_deseada]=tpoly(q0,qf,t);

% Inicialiazaciones (si se requieren)
ep = zeros(1,length(Q_deseada));
torque = zeros(1,length(Q_deseada));
q_real = zeros(1,length(Q_deseada));
qd_real = zeros(1,length(Q_deseada));
```

Figura 21: Selección de la articulación a controlar, trayectoria e inicializaciones

A continuación, se desarrolla el bucle de control donde se habilita la sincronización y comienza la simulación. Se leen de *CoppeliaSim* la velocidad articular q_d, la posición articular q y el par articular, este último no empleado para el control.

Por tanto, se calcula el error como la posición deseada (trayectoria generada con tpoly) menos la posición real (lectura de la simulación en *CoppeliaSim)*, este error se multiplica por la ganancia *K*, dicho producto es el término que aporta el controlador, el cual se suma a la velocidad deseada (también generada con tpoly) que se envía al UR3, como se muestra en la Figura 22:

```
for i=1:length(Q_deseada)

    %Leer q de vrep
    [~,qd_real(i)] = vrep.simxGetObjectFloatParameter(clientID, ur3,2012,vrep.simx_opmode_oneshot_wait);
    [~,q_real(i)] = vrep.simxGetJointPosition(clientID, ur3, vrep.simx_opmode_oneshot_wait);
    [~,torque(i)] = vrep.simxGetJointForce(clientID, ur3, vrep.simx_opmode_oneshot_wait);

    ep(i) = Q_deseada(i)-q_real(i);
    qd_PID = ep(i)*Kp;
    qd_robot = qd_PID + Qd_deseada(i);

    %Mandar qd_robot a vrep
    vrep.simxSetJointTargetVelocity(clientID,ur3,qd_robot,vrep.simx_opmode_oneshot);
    %Enviar de forma sincrona los ultimos valores
    vrep.simxSynchronousTrigger(clientID);

end

vrep.simxPauseSimulation(clientID,vrep.simx_opmode_oneshot);
```

Figura 22: Controlador cinemático articular simulado

Mediante el comando *.simXSetJointTargetVelocity*, se envía desde MATLAB la velocidad que el motor de la articulación de CoppeliaSim deberá aplicar. De este modo, al finalizar el bucle de control que tiene la longitud de la trayectoria deseada, el robot estará situado en la posición que debe estar.

Para finalizar, se cierra la conexión con *CoppeliaSim* y se grafican las trayectorias deseadas generadas con *tpoly* y las trayectorias reales (leídas de CoppeliaSim), así como el error, como se corroborará en capítulo 4 de resultados.

3.5. Control Cinemático Cartesiano Simulado en el UR3

De forma similar que para el controlador cinemático articular, se deberá iniciar la API tanto en CoppeliaSim como en MATLAB, como se muestra en las Figuras 19 y 20, a diferencia de que el robot no se podrá colocar en su posición de reposo, ya que un incremento en el eje z le haría salir de su espacio de trabajo lo que daría problemas de singularidades, luego se coloca en una posición articular específica: q = [0.87,0,-pi/2,-pi/2,0.35,0] como se observa en la Figura 23:

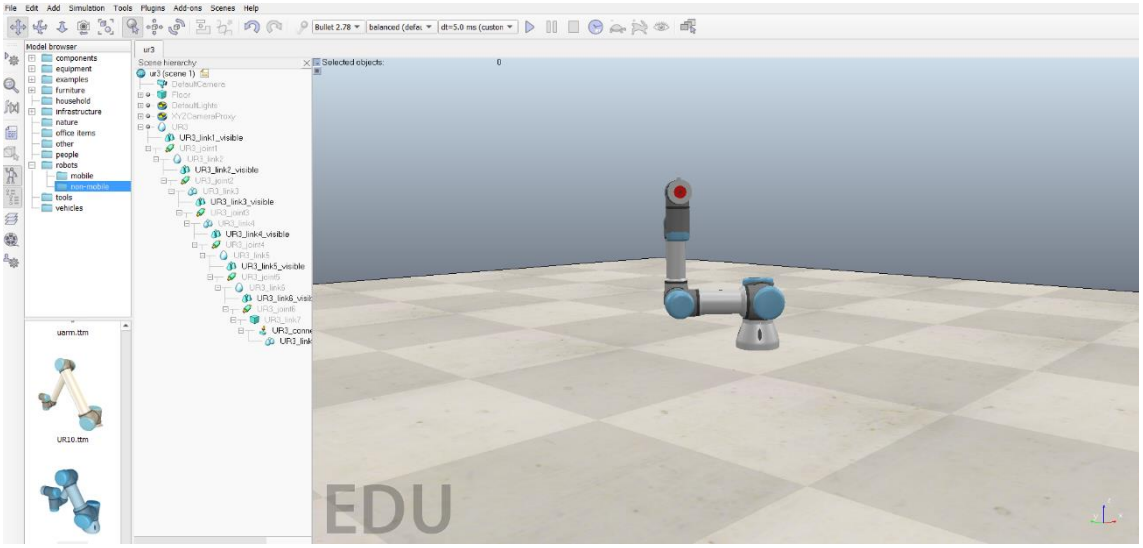


Figura 23: Posición de partida para el controlador cinemático Cartesiano simulado

Así, se logra que el robot no salga de su espacio de trabajo y se consiguen evitar problemas de singularidades ya que para realizar un desplazamiento en el eje z, luego en el eje y, por último, en el eje x a partir de la posición en la que se ha colocado, no se alinean más de 2 ejes.

Luego tras haber inicializado la comunicación en ambos programas y situado el UR3 en una configuración lejana de posiciones singulares, se procede a ejecutar mdl_ur3.m, *script* que crea el modelo del UR3 a partir del cual se pueden hacer cálculos cinemáticos, ya que dicho archivo crea un modelo cinemático del manipulador siguiendo el convenio DH, seguidamente, se obtienen los *handles* (identificadores) de todas las articulaciones, incluyendo los sistemas de referencia del extremo y de la base como se observa en la Figura 24:

```
% Recibimos datos del UR3

[~,ur1]=vrep.simxGetObjectHandle(clientID,'UR3_joint1',vrep.simx_opmode_oneshot_wait);
[~,ur2]=vrep.simxGetObjectHandle(clientID,'UR3_joint2',vrep.simx_opmode_oneshot_wait);
[~,ur_3]=vrep.simxGetObjectHandle(clientID,'UR3_joint3',vrep.simx_opmode_oneshot_wait);
[~,ur4]=vrep.simxGetObjectHandle(clientID,'UR3_joint4',vrep.simx_opmode_oneshot_wait);
[~,ur5]=vrep.simxGetObjectHandle(clientID,'UR3_joint5',vrep.simx_opmode_oneshot_wait);
[~,ur6]=vrep.simxGetObjectHandle(clientID,'UR3_joint6',vrep.simx_opmode_oneshot_wait);
[~,ur_tool]=vrep.simxGetObjectHandle(clientID,'UR3_connection',vrep.simx_opmode_oneshot_wait);
[~,ur_base]=vrep.simxGetObjectHandle(clientID,'UR3',vrep.simx_opmode_oneshot_wait);

[~,pos] =vrep.simxGetObjectPosition(clientID, ur_tool, ur_base, vrep.simx_opmode_oneshot_wait);
[~,ori] =vrep.simxGetObjectOrientation(clientID, ur_tool, ur_base, vrep.simx_opmode_oneshot_wait);
```

Figura 24: Obtención identificadores para el controlador cinemático Cartesiano

Seguidamente, se procede a generar la trayectoria como se ha explicado en el capítulo 3.3 para el controlador cinemático Cartesiano, se establece la ganancia *K*, en este caso a 0.03 y comienza el bucle de control. Lo primero es habilitar la sincronización y comenzar la simulación, luego, se leen de CoppeliaSim la posición y velocidad del extremo, posiciones articulares (6 articulaciones), par articular y orientación del extremo, estos 2 últimos no empleados para el control como se muestra en la Figura 25:

```
% Bucle de control
% Ganancia
K=0.03;

% Habilita la sincronización y comienza la simulacion
vrep.simxStartSimulation(clientID,vrep.simx_opmode_oneshot_wait);
vrep.simxSynchronous(clientID,1);
vrep.simxSynchronousTrigger(clientID);

for i=1:max(size(sdZ))
    [~,pos] =vrep.simxGetObjectPosition(clientID, ur_tool, ur_base, vrep.simx_opmode_oneshot_wait);
    [~,ori] =vrep.simxGetObjectOrientation(clientID, ur_tool, ur_base, vrep.simx_opmode_oneshot_wait);
    [~,q1] =vrep.simxGetJointPosition(clientID, ur1, vrep.simx_opmode_oneshot_wait);
    [~,q2] =vrep.simxGetJointPosition(clientID, ur2, vrep.simx_opmode_oneshot_wait);
    [~,q3] =vrep.simxGetJointPosition(clientID, ur_3, vrep.simx_opmode_oneshot_wait);
    [~,q4] =vrep.simxGetJointPosition(clientID, ur4, vrep.simx_opmode_oneshot_wait);
    [~,q5] =vrep.simxGetJointPosition(clientID, ur5, vrep.simx_opmode_oneshot_wait);
    [~,q6] =vrep.simxGetJointPosition(clientID, ur6, vrep.simx_opmode_oneshot_wait);
    [~,tau(1,i)] =vrep.simxGetJointForce(clientID, ur1, vrep.simx_opmode_oneshot_wait);
    [~,tau(2,i)] =vrep.simxGetJointForce(clientID, ur2, vrep.simx_opmode_oneshot_wait);
    [~,tau(3,i)] =vrep.simxGetJointForce(clientID, ur_3, vrep.simx_opmode_oneshot_wait);
    [~,tau(4,i)] =vrep.simxGetJointForce(clientID, ur4, vrep.simx_opmode_oneshot_wait);
    [~,tau(5,i)] =vrep.simxGetJointForce(clientID, ur5, vrep.simx_opmode_oneshot_wait);
    [~,tau(6,i)] =vrep.simxGetJointForce(clientID, ur6, vrep.simx_opmode_oneshot_wait);
    [~,vel_lineal] = vrep.simxGetObjectVelocity(clientID, ur_tool,vrep.simx_opmode_oneshot_wait);
```

Figura 25: Lectura parámetros para el controlador cinemático Cartesiano simulado

Tras haber leído los parámetros de CoppeliaSim, podemos desarrollar el controlador como se muestra en la Figura 26:

```
vel_real(1,i)=vel_lineal(1);
vel_real(2,i)=vel_lineal(2);
vel_real(3,i)=vel_lineal(3);
realposX(i)=pos(1);
realposY(i)=pos(2);
realposZ(i)=pos(3);
pdotX=sdX(i)+K*(sX(i)-pos(1));
pdotY=sdY(i)+K*(sY(i)-pos(2));
pdotZ=sdZ(i)+K*(sZ(i)-pos(3));
q=[double(q1) double(q2) double(q3) double(q4) double(q5) double(q6)];
jacobiana = ur3.jacob0(q);


qdot=inv(jacobiana)*[pdotX pdotY pdotZ 0 0 0]';
```

Figura 26: Código para el controlador cinemático Cartesiano simulado

La metodología consiste en almacenar las posiciones y velocidades del extremo en xyz lo que corresponde con *realpos* y *vel_lineal*, de forma que se desarrolla una expresión para cada eje, *pdot*. Dicha expresión consiste en la velocidad del extremo deseada, más la ganancia K multiplicada por el error en posición. Seguidamente se almacenan las posiciones articulares en un vector de forma que se pueda calcular la matriz Jacobiana.

Luego, las velocidades articulares a enviar para conseguir los desplazamientos del extremo se obtienen multiplicado la inversa de la matriz Jacobiana por el vector de controladores en posición (1 para cada eje) como se muestra en la Figura 27, sin tener en cuenta la orientación del extremo ya que no es necesario y ralentizaría los cálculos.

```
    qdot=inv(jacobiana)*[pdotX pdotY pdotZ 0 0 0]';

    realvel(:,i)=qdot;

    returnCode=vrep.simxSetJointTargetVelocity(clientID,ur1,qdot(1),vrep.simx_opmode_oneshot);
    returnCode=vrep.simxSetJointTargetVelocity(clientID,ur2,qdot(2),vrep.simx_opmode_oneshot);
    returnCode=vrep.simxSetJointTargetVelocity(clientID,ur_3,qdot(3),vrep.simx_opmode_oneshot);
    returnCode=vrep.simxSetJointTargetVelocity(clientID,ur4,qdot(4),vrep.simx_opmode_oneshot);
    returnCode=vrep.simxSetJointTargetVelocity(clientID,ur5,qdot(5),vrep.simx_opmode_oneshot);
    returnCode=vrep.simxSetJointTargetVelocity(clientID,ur6,qdot(6),vrep.simx_opmode_oneshot);
    % Contador termina
    vrep.simxSynchronousTrigger(clientID);
    % Contador empieza
end

vrep.simxPauseSimulation(clientID,vrep.simx_opmode_oneshot);
```

Figura 27: Velocidades articulares en el controlador cinemático Cartesiano simulado

Por tanto, una vez terminado el bucle de control, finaliza la simulación (se cierra la conexión con CoppeliaSim) y se grafican los resultados, que muestran que el extremo del robot debe haber sido capaz de seguir las referencias generadas, lo que se corroborará en el capítulo 4 de resultados.

3.6. Comunicación TCP/IP vía socket

El modelo TCP/IP es usado para comunicaciones en redes y como todo protocolo, describe un conjunto de guías generales de operación para permitir que un equipo pueda comunicarse en una red. TCP/IP provee conectividad de extremo a extremo especificando cómo los datos deberían ser formateados, direccionados, transmitidos, enrutados y recibidos por el destinatario.

De este modo, empleando el puerto TCP/IP disponible en la caja controladora del UR3, se necesita de un *router*, al que también deberá estar conectado el pc. Así, se consigue estar conectado a una red *ethernet*, donde se encuentran el router, el UR3 y el pc desde donde se va a ejecutar el controlador.

Así, una vez realizado el conexionado, se estará conectado a una red donde se podrá tanto enviar como recibir información entre los clientes y el servidor mediante un *socket* empleando la dirección IP del *router* y el puerto del robot que permite envío y recepción de datos.

3.7. Problemas de comunicaciones

Ya que los controladores para las simulaciones estaban diseñados en MATLAB, la finalidad del proyecto era extrapolarlos al robot real, de modo que ambos fueran gestionados por el mismo programa. Cuando se trabaja con el robot real, a diferencia de la simulación, no se puede leer solo la información que se requiere, sino un paquete que contiene todos los parámetros del robot.

El tamaño de dicho paquete varía en función de la versión del controlador que esté instalada en el Polyscope del UR3. En el laboratorio, se disponía de la versión 3.14. UR proporciona un archivo separado por comas *client_interfaces_1.1.3* (.csv) donde indica los tamaños del paquete que envía por el puerto de comunicaciones externas en función de la versión controladora.

Para la versión de Polyscope 3.14, el tamaño del paquete es de 1140 bytes como se muestra en la Tabla 2:

Meaning	Type	Number of values	Size in bytes	Gnuplot col.	Notes
Message Size	integer	1	4		Total message length in bytes
Time	double	1	8	1	Time elapsed since the controller was started
q target	double	6	48	2 - 7	Target joint positions
qd target	double	6	48	8 - 13	Target joint velocities
qdd	double	6	48	14 - 19	Target joint accelerations
I target	double	6	48	20 - 25	Target joint currents
M target	double	6	48	26 - 31	Target joint moments (torques)
q actual	double	6	48	32 - 37	Actual joint positions
qd actual	double	6	48	38 - 43	Actual joint velocities
I actual	double	6	48	44 - 49	Actual joint currents
I control	double	6	48	50 - 55	Joint control currents
Tool	double	6	48	56 - 61	Actual Cartesian coordinates of the tool: (x,y,z,rx,ry,rz), where rx, ry and rz is a rotation vector representation of the tool orientation
TCP	double	6	48	62 - 67	Actual speed of the tool given in Cartesian coordinates
TCP force	double	6	48	68 - 73	Generalised forces in the TCP
Tool	double	6	48	74 - 79	Target Cartesian coordinates of the tool: (x,y,z,rx,ry,rz), where rx, ry and rz is a rotation vector representation of the tool orientation
TCP	double	6	48	80 - 85	Target speed of the tool given in Cartesian coordinates
Digital input bits	double	1	8	86	Current state of the digital inputs. NOTE: these are bits encoded as int64_t, e.g. a value of 5 corresponds to bit 0 and bit 2 set high
Motor	double	6	48	87 - 92	Temperature of each joint in degrees celsius
Controller Timer	double	1	8	93	Controller realtime thread execution time
Test value	double	1	8	94	A value used by Universal Robots software only
Robot Mode	double	1	8	95	Robot mode RobotMod
Joint	double	6	48	96-101	Joint control modes ControlModes
Safety Mode	double	1	8	102	Safety mode SafetyMo
	double	6	48	103 - 108	Used by Universal Robots software only
Tool	double	3	24	109 - 111	Tool x,y and z accelerometer values (software version 1.7)
	double	6	48	112 - 117	Used by Universal Robots software only
Speed scaling	double	1	8	118	Speed scaling of the trajectory limiter
Linear momentum norm	double	1	8	119	Norm of Cartesian linear momentum
	double	1	8	120	Used by Universal Robots software only
	double	1	8	121	Used by Universal Robots software only
V main	double	1	8	122	Masterboard: Main voltage
V robot	double	1	8	123	Masterboard: Robot voltage (48V)
I robot	double	1	8	124	Masterboard: Robot current
V actual	double	6	48	125 - 130	Actual joint voltages
Digital outputs	double	1	8	131	Digital outputs
Program state	double	1	8	132	Program state
Elbow	double	3	24	133 - 135	Elbow position
Elbow	double	3	24	136 -138	Elbow velocity
Safety	double	1	8	139	Safety status SafetyStat
	double	1	8	140	Used by Universal Robots software only
	double	1	8	141	Used by Universal Robots software only
	double	1	8	142	Used by Universal Robots software only
TOTAL		143	1140		143 values in a 1140 byte package

Tabla 2: Tamaño del paquete de datos que envía el UR3 cada 8ms. Fuente: [6]

Como se observa en la tabla 2, existen 143 valores distintos los cuales deben almacenarse en el orden correcto ya que se necesita acceder a ellos para obtener las posiciones articulares, por ejemplo.

Luego, se creó una función en *MATLAB* encargada de leer la información del robot y almacenarla en su lugar correspondiente. El problema surge cuando se crea un bucle de lectura para verificar que se puede leer correctamente la información del UR3.

En principio debería ser a tiempo real, es decir, que, si desde el *FlexPendant* se desplaza la articulación de la base 10 grados, la lectura de la posición de la base del UR3 que se debería estar recibiendo en MATLAB debería variar 10 grados, a la iteración siguiente, ya que el robot está enviando información constantemente cada 8ms, lo cual no ocurrió así.

Se intentó calcular la trayectoria deseada en función del tiempo de lectura del paquete, modificando la función *tpoly* de forma que no generase la trayectoria en función de un periodo de muestreo fijo y hubiese problemas de sincronización, ya que cada vez el bucle de control tenía una duración, y devolviese los coeficientes del interpolador quíntico para evaluarlos en función de la duración de la lectura. Como estrategia es correcta, ya que la única diferencia es que la trayectoria deseada no tiene un periodo de muestreo fijo, pero así tampoco se logró realizar el control sobre el robot real desde *MATLAB.*

El tiempo que tarda *MATLAB* en reaccionar a ese movimiento de la base oscila entre 3 y 5 segundos, por motivos ajenos al control y que se desconocen, lo que introduce un retardo inviable a la hora de realizar un control, luego, surgió la necesidad de encontrar una solución.

Por tanto, el objetivo alcanzado este punto era lograr leer a tiempo real la información proporcionada por el UR3, de forma que aplicar un control externo resultase factible y realizable.

La solución fue emplear el lenguaje de programación Python desde VisualStudio Code, de forma que se volvió a realizar un bucle de lectura teniendo en cuenta el tamaño del paquete especificado por UR para la versión de Polyscope 3.14.

A diferencia de MATLAB, los cambios en la lectura eran instantáneos, es decir, a cualquier pequeño desplazamiento a nivel articular desde el *FlexPendant*, el bucle de lectura se percataba del cambio a la iteración siguiente lo que sí que permite aplicar un control externo.

Una vez solventado el problema de la lectura, surgieron algunos problemas más, desde *MATLAB*, la trayectoria deseada se genera con la función *tpoly* disponible en la *RoboticsToolbox for MATLAB* de Peter Corke, de la cual no se dispone en Python y se tuvo que replicar. Además, esta *toolbox* proporciona los modelos cinemáticos de los principales robots manipuladores como es el UR3, necesarios para realizar cálculos como la matriz Jacobiana. Por suerte, existe una adaptación de esta *toolbox* a modo de librería para *Python*, de forma que este problema no tuvo más cabida.

En el capítulo 3.8 se expone la réplica a *tpoly* extrapolada para *Python*, más que necesaria para poder generar las trayectorias correctamente.

3.8. Interpolador para la generación de la trayectoria real

Dado que la librería para *Python* de Peter Corke no incorpora el interpolador necesario para generar los perfiles de posición y velocidad requeridos, surgió la necesidad de adaptar tpoly para *Python*, de forma que se pudieran generar las trayectorias tanto para el controlador cinemático articular como para el Cartesiano.

Primero, se deben importar las librerías necesarias, como se observa en la Figura 28:

```
import numpy as np
import time
from matplotlib import pyplot as plt
```

Figura 28: Librerías necesarias para el interpolador en *Python*

A continuación, se traspone el vector de tiempos si es un escalar, si no, se deja como está y se asigna el tiempo final al valor máximo del vector de tiempos *t*, con velocidad inicial y final nulas. Seguidamente, se construye la matriz para obtener los coeficientes del polinomio empleando el método de mínimos cuadrados, véase la Figura 29:

```
    X = np.array([[0, 0, 0, 0, 0, 1],
                  [tf**5, tf**4, tf**3, tf**2, tf, 1],
                  [0, 0, 0, 0, 1, 0],
                  [5*tf**4, 4*tf**3, 3*tf**2, 2*tf, 1, 0],
                  [0, 0, 0, 2, 0, 0],
                  [20*tf**3, 12*tf**2, 6*tf, 2, 0, 0]])
```

Figura 29: Matriz de coeficientes del interpolador para *Python*

Dichos coeficientes, se obtienen de dividir la matriz anterior entre el vector que contiene el punto inicial y final de la trayectoria, con los demás parámetros de velocidad nulos. Luego, al obtener los coeficientes, habrá que derivarlos para obtener los relativos a velocidad y aceleración: *coeffs_d y coeffs_dd* como se muestra en la Figura 30. Por último, en el caso que se quisiera generar una trayectoria con un vector de tiempos fijo, la función puede devolver *p, pd y pdd,* pero interesa devolver también los coeficientes y calcular la trayectoria en función de la duración del bucle de lectura para evitar problemas de sincronización.

```
coeffs = np.array(np.transpose((np.linalg.lstsq(X, np.transpose([q0, qf,
0, 0, 0, 0]),rcond=None))))
coeffs_1 = coeffs[0]
coeffs_2 = coeffs_1[0:5]



coeffs_d = coeffs_2 * [5,4,3,2,1]
coeffs_3 = coeffs_d[0:4]
coeffs_dd = coeffs_3 * [4,3,2,1]

p = np.polyval(coeffs_1, t)
pd = np.polyval(coeffs_d, t)
pdd = np.polyval(coeffs_dd, t)
```

Figura 30: Cálculo de los coeficientes del interpolador para Python

Por último, quedaría llamar a la función, asignar los parámetros de posición inicial y final, vector de tiempo y velocidad inicial y final, de forma que el interpolador pueda generar el perfil que describen los parámetros introducidos.

Para una posición inicial igual a 0, final de 1.57 radianes (90 grados), periodo de muestreo de 0.008 entre 0 y 2 segundos y velocidades inicial y final nulas. Los perfiles que se obtienen son los siguientes:

```
qq0 = 0
qqf = 1.57
t1 = np.arange(0,2,0.008)
qqd0 = 0
qqdf = 0
q0,qf,t,qd0,qdf = tpoly(qq0,qqf,t1,qqd0,qqdf)
```

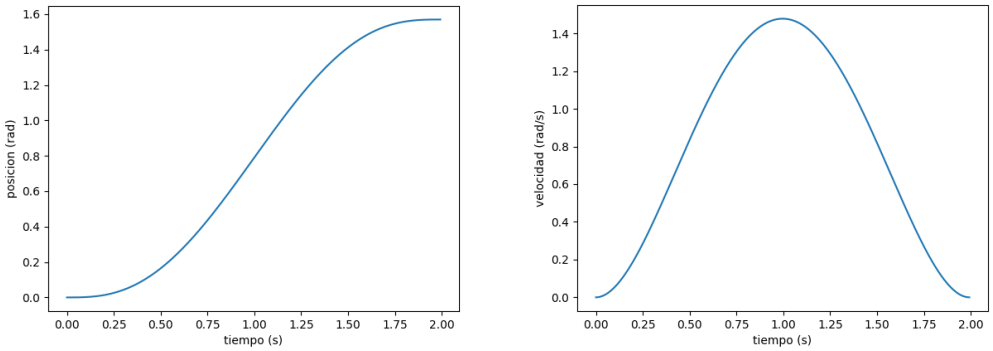


Figura 31: Trayectorias en Python para el controlador articular

Para el caso del Cartesiano, se deben generar las trayectorias de forma que se produzca un movimiento de 0.1 metros en el eje z de 0 a 2s, de 0.1m en y de 2 a 4 y 0.1m en x de 4 a 6s por ejemplo, de forma que las posiciones iniciales de los 3 ejes corresponden con la posición del extremo del robot en la posición: q = [0.87,0,-pi/2,-pi/2,0.35,0].

El código necesario para generar dicha trayectoria es el siguiente:

```
import time
import struct
import numpy as np
from matplotlib import pyplot as plt
from tpoly import tpoly


x_tcp = 0.00424
y_tcp = -0.28712
z_tcp = 0.45192


def trajectory():

    sX = []
    sY = []
    sZ = []
    sdX = []
    sdY = []
    sdZ = []

    t = np.arange(0,0.66,0.008)
    t_plot = np.arange(0,6-0.024,3*0.008)


    px, pdx, coeffs_1, coeffs_d1 = tpoly(x_tcp,x_tcp,t,0,0)
    py, pdy, coeffs_2, coeffs_d2 = tpoly(y_tcp,y_tcp,t,0,0)
    pz, pdz, coeffs_3, coeffs_d3= tpoly(z_tcp,z_tcp+0.1,t,0,0)



    p1x, pd1x, coeffs_11, coeffs_d11 = tpoly(x_tcp,x_tcp,t,0,0)
    p1y, pd1y, coeffs_22, coeffs_d22 = tpoly(y_tcp,y_tcp+0.1,t,0,0)
    p1z, pd1z, coeffs_33, coeffs_d33 = tpoly(z_tcp+0.1,z_tcp+0.1,t,0,0)


    p2x, pd2x, coeffs_111, coeffs_d111 = tpoly(x_tcp,x_tcp+0.1,t,0,0)
    p2y, pd2y, coeffs_222, coeffs_d222 = tpoly(y_tcp+0.1,y_tcp+0.1,t,0,0)
    p2z, pd2z, coeffs_333, coeffs_d333 = tpoly(z_tcp+0.1,z_tcp+0.1,t,0,0)
```

```
sX.extend(px)
sX.extend(p1x)
sX.extend(p2x)
sY.extend(py)
sY.extend(p1y)
sY.extend(p2y)
sZ.extend(pz)
sZ.extend(p1z)
sZ.extend(p2z)

sdX.extend(pdx)
sdX.extend(pd1x)
sdX.extend(pd2x)
sdY.extend(pdy)
sdY.extend(pd1y)
sdY.extend(pd2y)
sdZ.extend(pdz)
sdZ.extend(pd1z)
sdZ.extend(pd2z)
```

Donde al plotear los resultados, se obtienen los mismos perfiles que en el capítulo 3.3 como se muestra en la Figura 32.

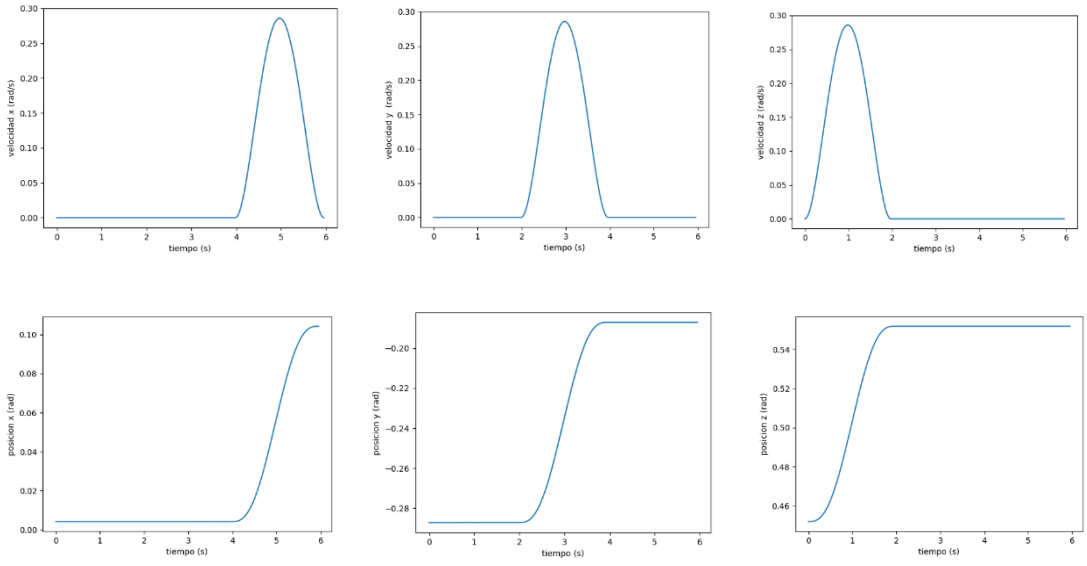


Figura 32: Trayectorias en Python para el controlador Cartesiano

3.9. Control Cinemático Articular sobre el robot real UR3

Primero que todo, se importan las librerías, se declara la dirección IP de la red y se especifica el puerto el cual se va a atacar con el socket. Seguidamente se llama a la función *tpoly* para obtener los coeficientes correspondientes a la trayectoria deseada. A continuación, se hacen las declaraciones pertinentes y se abre el socket, como se muestra en la Figura 33:

```
import socket
import time
import struct
import numpy as np
from matplotlib import pyplot as plt
from tpoly import tpoly

HOST = '192.168.2.2'
PORT_30003 = 30003
t = np.arange(0,2,0.008)

p, pd, coeffs_1, coeffs_d = tpoly(0,1.57,t,0,0)
print ("Starting Program")


qf = 1.57
q_real = []
pos = []
vel = []
error = []
qd_real = []
save_duration = []
qd_zeros = [0]*6
ep = []
Kp = 2.5
i = 0
flag = 1
start = time.time()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT_30003))
```

Figura 33: Inicializaciones y apertura socket controlador articular

Tras esto, se inicia un contador antes de acceder al bucle de control donde se procede a recibir el paquete de 1140 bytes que envía el UR3 por el puerto 30003 cada 8 milisegundos, desempaquetando para este controlador, las posiciones y velocidades articulares, una vez recibidas desempaquetadas, termina el contador y almacenamos el tiempo como se muestra en la Figura 34:

```
while(flag):

        packet_1 = s.recv(4)
        packet_2 = s.recv(8)
        packet_3 = s.recv(48)
        packet_4 = s.recv(48)
        packet_5 = s.recv(48)
        packet_6 = s.recv(48)
        packet_7 = s.recv(48)
        packet_8_1 = s.recv(8)
        packet_8_2 = s.recv(8)
        packet_8_3 = s.recv(8)
        packet_8_4 = s.recv(8)
        packet_8_5 = s.recv(8)
        packet_8_6 = s.recv(8)
        packet_9_1 = s.recv(8)
        packet_9_2 = s.recv(8)
        packet_9_3 = s.recv(8)
        packet_9_4 = s.recv(8)
        packet_9_5 = s.recv(8)
        packet_9_6 = s.recv(8)
        packets_extra = s.recv(792)

        q1 = str(packet_8_1)
        q1 = struct.unpack("!d", packet_8_1)[0]
        q2 = str(packet_8_2)
        q2 = struct.unpack("!d", packet_8_2)[0]
        q3 = str(packet_8_3)
        q3 = struct.unpack("!d", packet_8_3)[0]
        q4 = str(packet_8_4)
        q4 = struct.unpack("!d", packet_8_4)[0]
        q5 = str(packet_8_5)
        q5 = struct.unpack("!d", packet_8_5)[0]
        q6 = str(packet_8_6)
        q6 = struct.unpack("!d", packet_8_6)[0]
        q = [q1, q2, q3, q4, q5, q6]
```

```
        print ("q = ", q)
        q_real.append(q1)


        qd1 = str(packet_9_1)
        qd1 = struct.unpack("!d", packet_9_1)[0]
        qd2 = str(packet_9_2)
        qd2 = struct.unpack("!d", packet_9_2)[0]
        qd3 = str(packet_9_3)
        qd3 = struct.unpack("!d", packet_9_3)[0]
        qd4 = str(packet_9_4)
        qd4 = struct.unpack("!d", packet_9_4)[0]
        qd5 = str(packet_9_5)
        qd5 = struct.unpack("!d", packet_9_5)[0]
        qd6 = str(packet_9_6)
        qd6 = struct.unpack("!d", packet_9_6)[0]
        qd = [qd1, qd2, qd3, qd4, qd5, qd6]
        print ("qd = ", qd)
        qd_real.append(qd1)


        end = time.time()
        duration = end - start
        save_duration.append(duration)
```

Figura 34: Lectura del socket para el controlador articular

En este punto, se entra al apartado del controlador. Como se ha comentado anteriormente, las trayectorias deseadas en posición y en velocidad se van a calcular en función de la duración del contador que mide el tiempo de lectura, de este modo, se suprime cualquier tipo de problemas de sincronización ya que la trayectoria en cada iteración se va a generar en el momento adecuado.

Dado que en la trayectoria se ha especificado un giro de 90 grados en 2 segundos, se procede al controlador que se muestra en la Figura 35:

```
        if duration < 2:
            posicion = np.polyval(coeffs_1, duration)
            velocidad= np.polyval(coeffs_d, duration)
        else:
            posicion = qf
            velocidad = 0
            flag = 0

        pos.append(posicion)
        vel.append(velocidad)

        ep = posicion - q_real[i]
        qd_PID = ep*Kp
        qd_robot = qd_PID + velocidad
        error.append(ep)

        string = 'speedj([%f, 0, 0, 0, 0, 0], 5, 2)' % (qd_robot) + '\n'
        strcode = string.encode()
        s.send (strcode)

        i = i + 1
        print ("Program finish")
```

Figura 35: Controlador cinemático articular sobre el UR3 real

Así, si la duración es menor que el tiempo máximo para generar el movimiento, se calcula la trayectoria deseada en función de la duración de la lectura evaluando los coeficientes de tpoly con dicha duración. Si el tiempo excede de 2 segundos, el controlador debe haber terminado.

De forma que se calcula el error como la posición deseada (evaluando los coeficientes en función de la duración) menos la posición real (lectura del socket). Este error se multiplica por la ganancia y dicho término se suma a la velocidad de envío, la cual también se calcula en función de la duración del bucle. Por último, quedaría hacer el envío de velocidad y así funciona el bucle hasta que se alcance la posición deseada una vez finalizado el tiempo. Quedaría graficar los resultados los cuales se exponen en el capítulo 4.

3.10. Control Cinemático Cartesiano sobre el robot real UR3

Para este controlador, de igual forma que el articular, se importan las librerías necesarias y se declaran las inicializaciones pertinentes. A diferencia del controlador anterior, aquí se necesita hacer cálculos cinemáticos como la matriz Jacobiana, luego se importa de la roboticstoolbox el modelo con las convenciones DH para el manipulador UR3, de forma que se tengan las relaciones y transformaciones desde el sistema de referencia de la base del robot hasta el extremo.

Se necesita hacer una lectura previa al bucle de control para saber exactamente cual es la posición en xyz del efector final como se muestra en la Figura 36, con el objetivo de que la trayectoria comience en ese punto, ya que aquí no se va a mover una sola articulación, sino todo el conjunto que produzca los desplazamientos del extremo. A continuación, se generan las trayectorias entre 0 y 6 segundos, reseteando el controlador al cambiar de eje en el bucle de control con el fin de que todas las trayectorias se generen entre 0 y 6, ya que, si se llama a tpoly con un tiempo inicial distinto de 0, los perfiles no se crean correctamente, independientemente de que los desplazamientos en y & x se produzcan entre 6 y 12, 12 y 18 respectivamente.

```
import socket
import time
import struct
import numpy as np
from matplotlib import pyplot as plt
from tpoly import tpoly
import roboticstoolbox as rtb



save_duration = []
posxx = []
velxx = []
posyy = []
```

```
velyy = []
poszz = []
velzz = []
K = 1
i = 0
flag = 1
qf = 0.1
pos_X = []
vel_X =[]
pos_Y = []
vel_Y =[]
pos_Z =[]
vel_Z = []
robot = rtb.models.DH.UR3()

xtcp = 0.00424
ytcp = -0.28712
ztcp = 0.45192


HOST = '192.168.2.2' # The remote host
PORT_30003 = 30003

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    #s.settimeout(1)
s.connect((HOST, PORT_30003))

packet_1 = s.recv(4)
packet_2 = s.recv(8)
packet_3 = s.recv(48)
packet_4 = s.recv(48)
packet_5 = s.recv(48)
packet_6 = s.recv(48)
packet_7 = s.recv(48)
packet_8_1 = s.recv(8)
packet_8_2 = s.recv(8)
packet_8_3 = s.recv(8)
packet_8_4 = s.recv(8)
packet_8_5 = s.recv(8)
packet_8_6 = s.recv(8)
packet_9 = s.recv(48)
packet_10 = s.recv(48)
packet_11 = s.recv(48)
packet_12_1 = s.recv(8)
packet_12_2 = s.recv(8)
packet_12_3 = s.recv(8)
packet_12_456 = s.recv(24)
#packet_12 = s.recv(48)
```

```
packet_13_1 = s.recv(8)

packet_13_2 = s.recv(8)
packet_13_3 = s.recv(8)
packet_13_456 = s.recv(24)
packets_extra = s.recv(600)
s.close()


# POSICIONES ARTICULARES

q1 = str(packet_8_1)
q1 = struct.unpack("!d", packet_8_1)[0]
q2 = str(packet_8_2)
q2 = struct.unpack("!d", packet_8_2)[0]
q3 = str(packet_8_3)
q3 = struct.unpack("!d", packet_8_3)[0]
q4 = str(packet_8_4)
q4 = struct.unpack("!d", packet_8_4)[0]
q5 = str(packet_8_5)
q5 = struct.unpack("!d", packet_8_5)[0]
q6 = str(packet_8_6)
q6 = struct.unpack("!d", packet_8_6)[0]

# POSICION TCP

pos_x = str(packet_12_1)
pos_x = struct.unpack("!d", packet_12_1)[0]
pos_y = str(packet_12_2)
pos_y = struct.unpack("!d", packet_12_2)[0]
pos_z = str(packet_12_3)
pos_z = struct.unpack("!d", packet_12_3)[0]

t = np.arange(0,6,0.008)
pz, pdz, coeffs_1z, coeffs_dz= tpoly(ztcp,ztcp+qf,t,0,0)
py, pdy, coeffs_1y, coeffs_dy= tpoly(ytcp,ytcp+qf,t,0,0)
px, pdx, coeffs_1x, coeffs_dx= tpoly(xtcp,xtcp+qf,t,0,0)


s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    #s.settimeout(1)
s.connect((HOST, PORT_30003))

start = time.time()
```

Figura 36: Inicializaciones y lectura previa al control Cartesiano

Por tanto, una vez importadas las librerías, declarado las inicializaciones, tomada la primera lectura para saber la posición del extremo, se vuelve a abrir el s*ocket*, se inicializa el contador y comienza el bucle de control.

Se vuelve a tomar la lectura y si la duración del bucle es inferior a 6 segundos se debe estar ejecutando el movimiento en el eje *z*, donde se calcula la duración de la lectura y se evalúa la trayectoria para este eje, señalando, que en los ejes x e y, la posición del extremo corresponde a la tomada en la primera lectura fuera del bucle. Seguidamente, se calcula el término del controlador como se muestra en la Figura 37 y se almacenan los resultados en vectores para su posterior graficado.

```
    end = time.time()
    duration = end - start

    if duration < 6:
        interval = end - start
        posiciony = ytcp
        velocidady = 0
        posicionx = xtcp
        velocidadx = 0
        posicionz = np.polyval(coeffs_1z, interval)
        velocidadz= np.polyval(coeffs_dz, interval)
        pdotZ=velocidadz+K*posicionz-pos_z
        pdotY=velocidady+K*posiciony-pos_y
        pdotX=velocidadx+K*posicionx-pos_x
        poszz.append(posicionz)
        velzz.append(velocidadz)
        posyy.append(posiciony)
        velyy.append(velocidady)
        posxx.append(posicionx)
        velxx.append(velocidadx)
        pos_Z.append(pos_z)
        vel_Z.append(vel_z)
        pos_Y.append(pos_y)
        vel_Y.append(vel_y)
        pos_X.append(pos_x)
        vel_X.append(vel_x)
```

Figura 37: Fragmento controlador Cartesiano eje z

Se observa que el término del controlador se calcula como la velocidad deseada, calculada en función de la duración del bucle de lectura más la ganancia multiplicada por el error en posición.

Luego, se calcula la matriz Jacobiana a partir de las posiciones articulares leídas y se multiplica la inversa de esta matriz por un vector formado por los términos del controlador *qdot* como se muestra en la Figura 38:

```
    J = robot.jacob0(q)
    Jacobiana = np.around(J, 3)

    qdot = np.matmul(np.linalg.inv(Jacobiana),np.transpose([pdotX, pdotY, pdo
    tZ, 0, 0, 0]))
    qd1 = qdot[0]
    qd2 = qdot[1]
    qd3 = qdot[2]
    qd4 = qdot[3]
    qd5 = qdot[4]
    qd6 = qdot[5]


    string = 'speedj([%f, %f, %f, %f, %f, %f], 20, 0.2)' % (qd1,qd2,qd3,qd4,q
    d5,qd6) + '\n'
    strcode = string.encode()
    s.send (strcode)
```

Figura 38: Cálculo de Jacobiana y envío de velocidades articulares

Así, qdot es un vector 1x6 donde cada columna corresponde a la velocidad de cada articulación. De este modo, mediante el comando speedj del lenguaje URscript de UR, se envían por el socket las velocidades articulares que producen los movimientos del extremo.

Si la duración del bucle de lectura es mayor que 6, el desplazamiento en el eje z debe haber terminado y comenzará el movimiento en el eje y, para ello, como las trayectorias han sido generadas de 0 a 6, se tiene que resetear el contador con el objetivo de que los perfiles se generen correctamente, como se muestra en la Figura 39:

```
if duration > 6 and duration < 12:
        interval = end - (start+6)
        posicionz = ztcp+0.1
        velocidadz = 0
        posicionx = xtcp
        velocidadx = 0
        posiciony = np.polyval(coeffs_1y, interval)
        velocidady= np.polyval(coeffs_dy, interval)
        pdotZ=velocidadz+K*posicionz-pos_z
        pdotY=velocidady+K*posiciony-pos_y
        pdotX=velocidadx+K*posicionx-pos_x
        poszz.append(posicionz)
        velzz.append(velocidadz)
        posyy.append(posiciony)
        velyy.append(velocidady)
        posxx.append(posicionx)
        velxx.append(velocidadx)
        pos_Z.append(pos_z)
        vel_Z.append(vel_z)
        pos_Y.append(pos_y)
        vel_Y.append(vel_y)
        pos_X.append(pos_x)
        vel_X.append(vel_x)
```

Figura 39: Fragmento controlador Cartesiano eje y

De esta forma, se consigue que el movimiento se produzca entre 6 y 12 segundos, fijando la posición del efector final en z en la posición inicial mas 0.1 ya que el movimiento en ese eje ya se a producido a diferencia del eje x

Por tanto, igual que en la Figura 38, se calcula la matriz Jacobiana para poder obtener *qdot* de nuevo con el fragmento de controlador correspondiente y se hace de nuevo el envío de velocidad, hasta que el desplazamiento en *y* haya terminado, tras llegar a una duración de 12 segundos.

Luego, una vez alcanzado este punto se habrán producido los desplazamientos en los ejes *z* e *y*. La forma de proceder en el eje *x* es la misma, como se muestra en la Figura 40:

```
if duration > 12 and duration < 18:
        interval = end - (start+12)
        posicionz = ztcp+0.1
        velocidadz = 0
        posiciony = ytcp+0.1
        velocidady = 0
        posicionx = np.polyval(coeffs_1x, interval)
        velocidadx= np.polyval(coeffs_dx, interval)
        pdotZ=velocidadz+K*posicionz-pos_z
        pdotY=velocidady+K*posiciony-pos_y
        pdotX=velocidadx+K*posicionx-pos_x
        poszz.append(posicionz)
        velzz.append(velocidadz)
        posyy.append(posiciony)
        velyy.append(velocidady)
        posxx.append(posicionx)
        velxx.append(velocidadx)
        pos_Z.append(pos_z)
        vel_Z.append(vel_z)
        pos_Y.append(pos_y)
        vel_Y.append(vel_y)
        pos_X.append(pos_x)
        vel_X.append(vel_x)
```

Figura 40: Fragmento controlador Cartesiano eje x

Para finalizar, si la duración es mayor que 18 se deben haber producido por completo los desplazamientos en *xyz*, ya que se habrá completado el tiempo de la trayectoria y se dejan de generar nuevos perfiles para más desplazamientos.

Por tanto, para salir del bucle *while*, se establece que si después del último envío de velocidad, el tiempo excede los 18 segundos, la variable *flag* pasa a ser cero y termina el bucle de control cerrando el socket como se muestra en la Figura 41:

```
if duration > 18:
        flag  = 0

i = i+1
```

Figura 41: Salida del bucle del controlador Cartesiano

Obsérvese que se ha introducido un contador, que aumenta una unidad a cada iteración con el objetivo de tener un vector de tiempos válido para graficar.

Por último, queda graficar los resultados que se muestran en el capítulo 4.

4. Resultados

En el presente capítulo se mostrarán los resultados obtenidos tras la experimentación realizada, haciendo uso de los controladores cinemáticos implementados, mostrados en el capítulo 3.

Primeramente, se mostrarán los resultados obtenidos a nivel de simulación, donde los controladores articular y Cartesiano están implementados en MATLAB y el UR3 se encuentra en una escena de CoppeliaSim, de modo que la comunicación entre los programas se realiza a través de un *socket.*

Tras la primera parte de simulación, se expondrán los resultados alcanzados con el robot UR3 físico. Los controladores están implementados en *Python* y la conexión con el robot real se realiza también a través de un *socket*, empleando un router y el puerto TCP/IP del robot como esclavo, siendo el maestro el pc desde el cual se ejecute el controlador.

Por último, se analizarán los gráficos de forma que se pueda realizar una comparativa entre los resultados que se obtienen a nivel simulado frente a la realidad, justificando tanto coincidencias como diferencias.

4.1. Resultados Controlador Cinemático Articular simulado

En el presente apartado, se muestra una parte del conjunto de pruebas realizadas en nivel simulado con el controlador cinemático articular para el UR3 entre *MATLAB* y *CoppeliaSim.*

En primer lugar, se mostrará la escena, donde el UR3 está en la posición de reposo, con todas sus coordenadas articulares a cero como se muestra en la Figura 42. La articulación que se desea controlar es la base, donde se le demanda un giro de 90 grados, el cual se efectúa gracias a la generación de la trayectoria y el controlador que minimiza el error. Cabe destacar que este control es aplicable a todas las articulaciones, aunque solo se muestren resultados de la primera articulación. Se han empleado 2 vectores de tiempos para lograr el movimiento, 2 y 10 segundos como se muestra más adelante. El efecto de la variación de la ganancia se muestra en los resultados a nivel real, ya que la simulación permite corroborar que era viable el diseño del controlador real

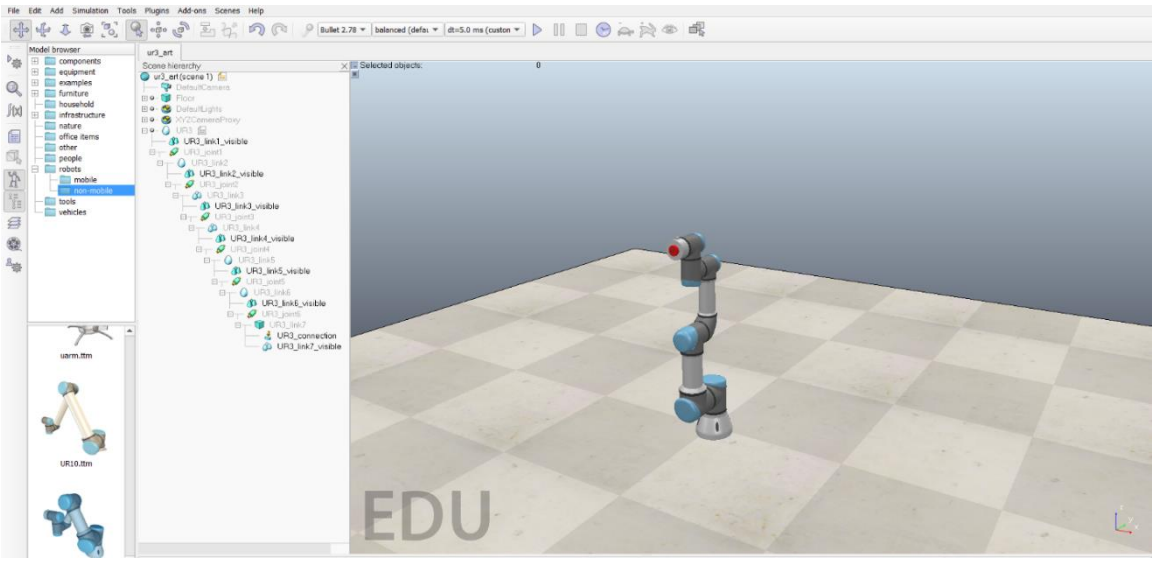


Figura 42: Posición inicial controlador articular simulado

Luego, los resultados que se han obtenido para un valor de ganancia Kp = 20 para los vectores de tiempo de 2 y 10 segundos son los siguientes:

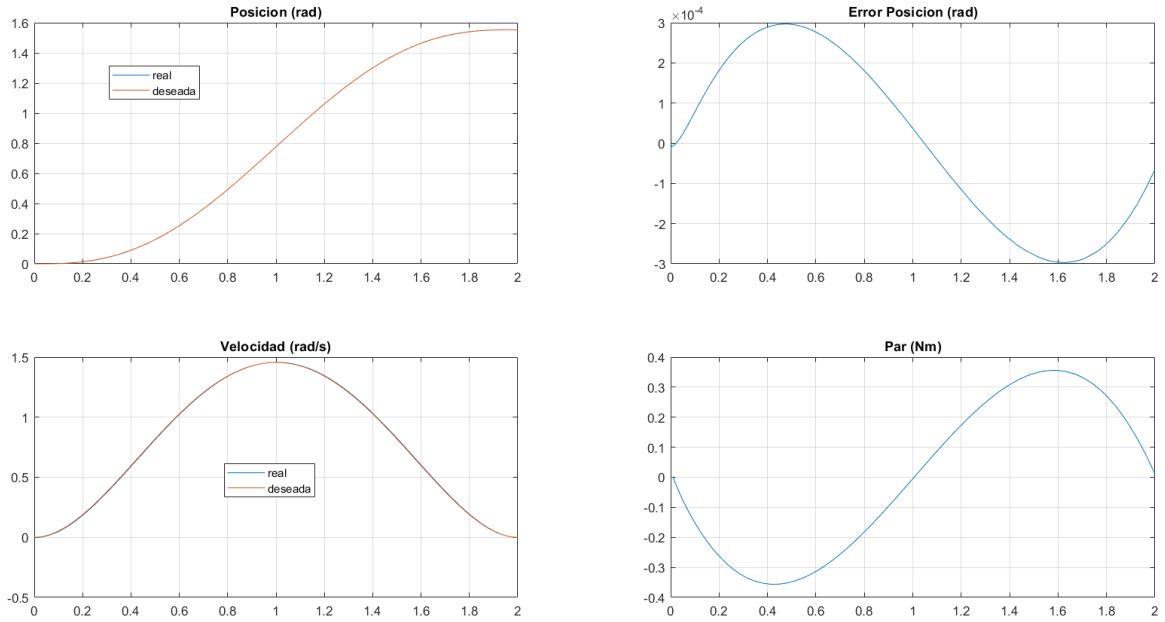


Figura 43: Resultados controlador articular simulado, Kp = 20 con t = 2s

Se puede observar que el seguimiento de la referencia tanto en posición como en velocidad es prácticamente perfecto, el error en posición máximo es de 3 x 10^{-4} radianes y 0.05 rad/s en velocidad. Se comprobó además que este tipo de controlador no demanda un par articular lo suficientemente alto como para que el robot no pudiera lograr el movimiento, el valor es entorno a los 0.4 Nm.

A continuación, se muestran los resultados obtenidos con la misma ganancia que el controlador anterior, pero con cinco veces más tiempo para realizar el movimiento, es decir, 10 segundos como se observa en la Figura 44:

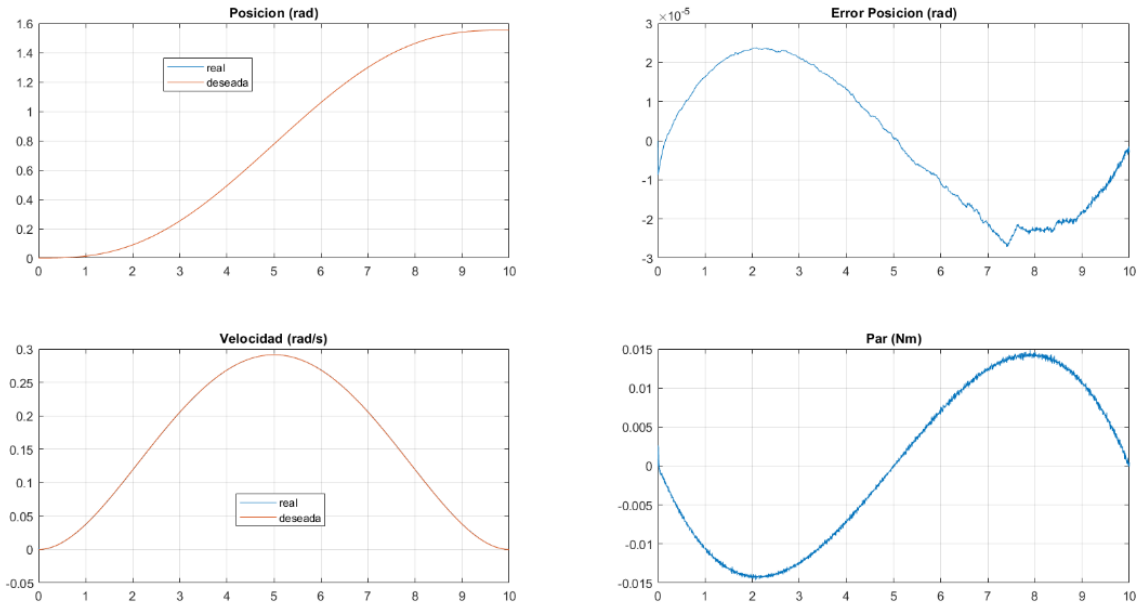


Figura 44: Resultados controlador articular simulado, Kp = 20con t = 10 s

El valor de la ganancia *Kp* es la misma en las 2 pruebas mostradas para el controlador cinemático articular simulado, por esta razón, el hecho de que el robot tenga que recorrer la misma trayectoria, pero en el segundo caso con 5 veces más tiempo hace que el error máximo en posición se vea reducido de 3 x 10^{-4} a 3 x 10^{-5} rad, lo que es una reducción grande teniendo en cuenta los órdenes de magnitud del error.

En cuanto a velocidad, en este segundo caso se observa que ha pasado de 1.5 rad/s (máxima) a 0.3, también por la razón de haber tenido más tiempo para recorrer la trayectoria y como el robot tiene que girar en ambos casos también noventa grados, la velocidad por obligación debe verse reducida como se observa comparando las Figuras 43 y 44.

Por consecuencia, el par articular en el segundo caso se ve reducido del orden de 25 veces, también porque la trayectoria se ejecuta en más tiempo.

Las pequeñas interferencias que se observan en los gráficos de error en posición y par se deben a que, para la ganancia especificada, el tiempo para la trayectoria es demasiado largo, ya que los errores obtenidos en la primera prueba son más que válidos para la mayoría de las aplicaciones industriales.

Tras el seguimiento de la trayectoria, en ambos casos el robot completa el giro de 90 grados como se corrobora en la Figura 45.

Cabe destacar que no solo se pueden generar trayectorias que se traduzcan en giros de 90 grados, sino que se puede trazar cualquiera en el rango de 0 a 360 grados.

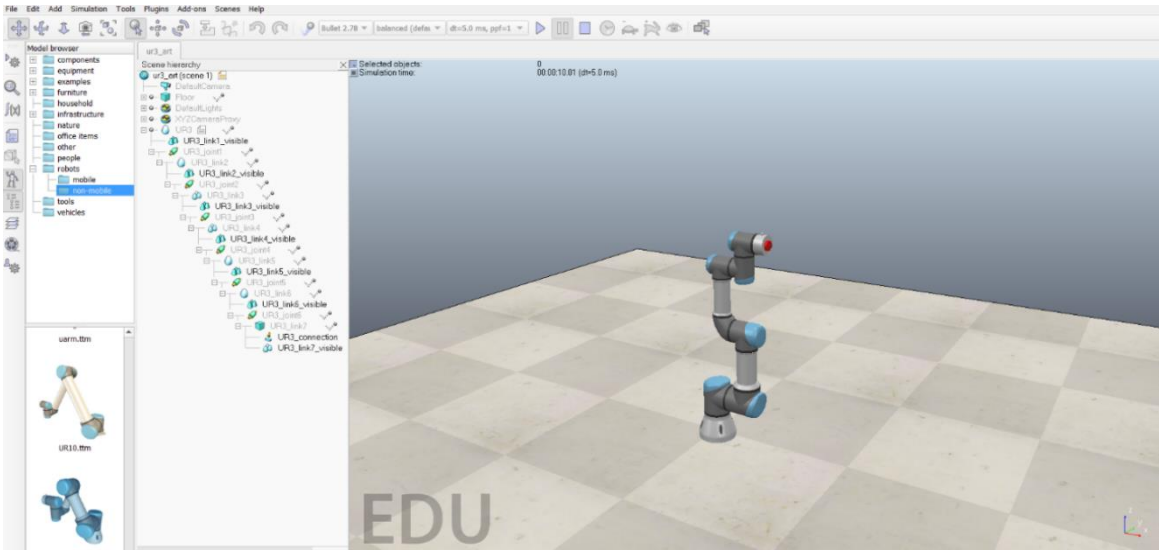


Figura 45: Posición final tras la trayectoria articular

Donde se puede verificar que a diferencia de la Figura 42, la articulación de la base se ve rotada 90 grados debido al correcto seguimiento de la trayectoria.

Se muestra una captura de la simulación de 2 segundos ya que la posición que se alcanza con la de 10 segundos es prácticamente la misma al nivel que aprecia los píxeles el ojo humano.

A continuación, se exponen los resultados obtenidos para el controlador cinemático Cartesiano a nivel también de simulación.

4.2. Resultados Controlador Cinemático Cartesiano simulado

En el presente apartado, se comparten los resultados alcanzados a nivel simulado con el controlador cinemático Cartesiano para el UR3. Se plantea una trayectoria del tipo mencionado en el capítulo 3.3 (posición inicial del efector final más 0.1m en los 3 ejes) de forma que se pretende realizar un desplazamiento en el eje *z* entre 0 y 2 segundos, en el eje *y* entre 2 y 4 segundos *y* por último en el eje x entre 4 y 6 segundos. El valor de ganancia *Kp* empleado es de 0.03 ya que valores superiores a esta cifra desestabilizaban el sistema. El objetivo de esta simulación ha sido de nuevo corroborar que es posible implementar el controlador a nivel simulado para luego implementarlo en el robot real.

Como se ha comentado en capítulos anteriores, para este tipo de controlador existen problemas de singularidades debido a que la matriz Jacobiana se calcula en todas las iteraciones del bucle de control de forma que para llevar a cabo este planteamiento el robot debe situarse en una posición lejana al alineamiento de 2 o más ejes como se muestra en la Figura 46.

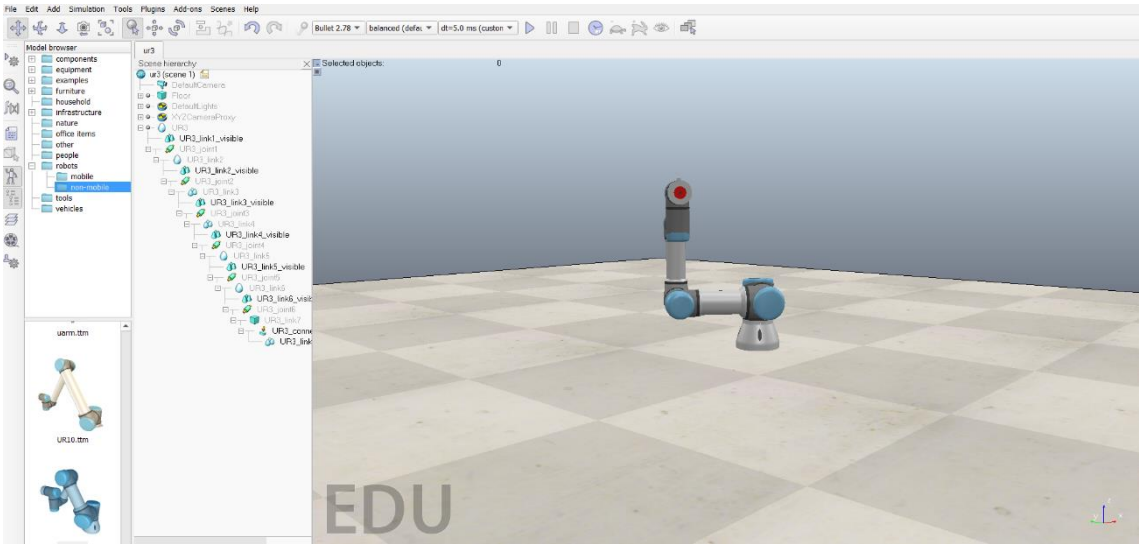


Figura 46: Posición inicial UR3 control Cartesiano simulado

Luego, los resultados obtenidos a nivel de posición son los siguientes:

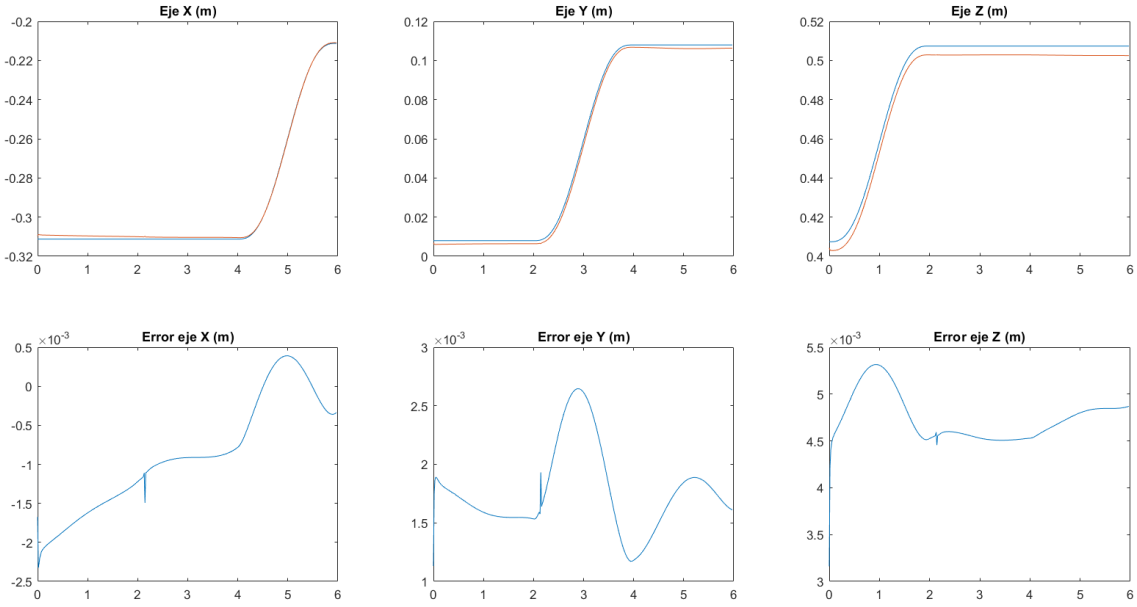


Figura 47: Evolución posición efector final controlador Cartesiano simulado

Como se puede observar, el seguimiento de las referencias en posición es correcto, en los tres ejes se consigue generar la trayectoria correctamente y el robot es capaz de seguirla como se corrobora con los gráficos de error, ya que su oren de magnitud es prácticamente despreciable. En los 3 ejes de observa como la trayectoria parte de la posición inicial del efector final para la posición mostrada en la Figura 46 y se observa que, transcurridos los intervalos de tiempo pertinentes, su posición se ve aumentada en 0.1 metros. Es cierto que, de los 3 ejes, el que mayor error posee es el eje z ya que inicialmente el robot está parado.

Los resultados a nivel de velocidad se muestran en la Figura 48:

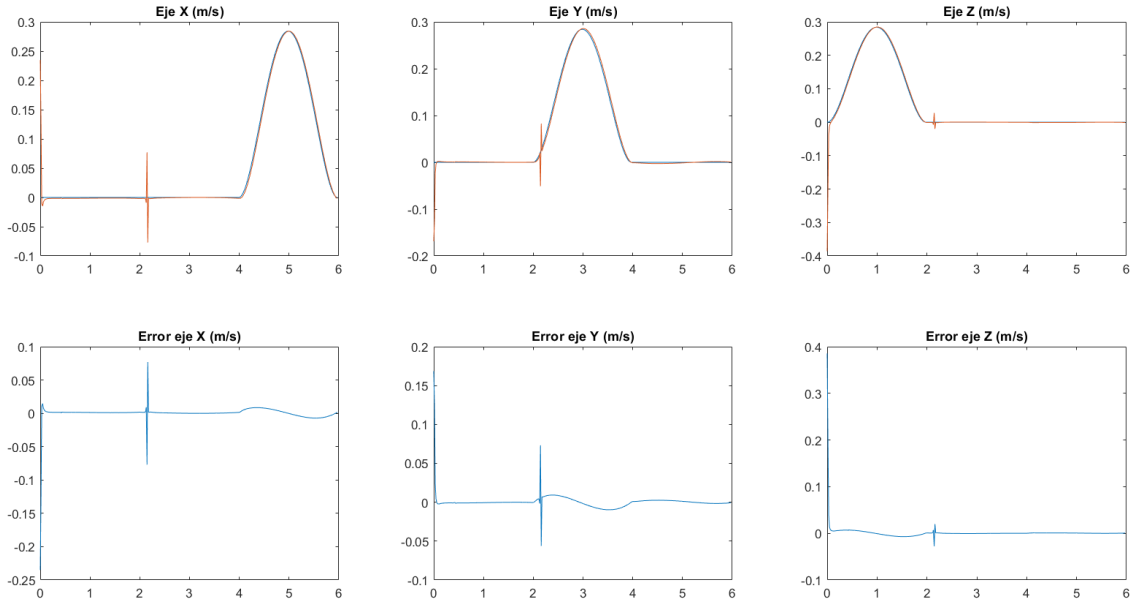


Figura 48: Evolución velocidad efector final controlador Cartesiano simulado

Como las referencias de posición se siguen bien, debe ocurrir lo mismo con las de velocidad, donde el error es completamente nulo en prácticamente todos los instantes, a diferencia del segundo 2 donde se observa un pico en la transición de movimiento del eje z al eje y que se debe a parámetros no modelados del simulador, por lo tanto, inevitables y que también se reflejan en los gráficos de velocidad y pares articulares que describen la trayectoria seguida como se muestra en la Figura 49:

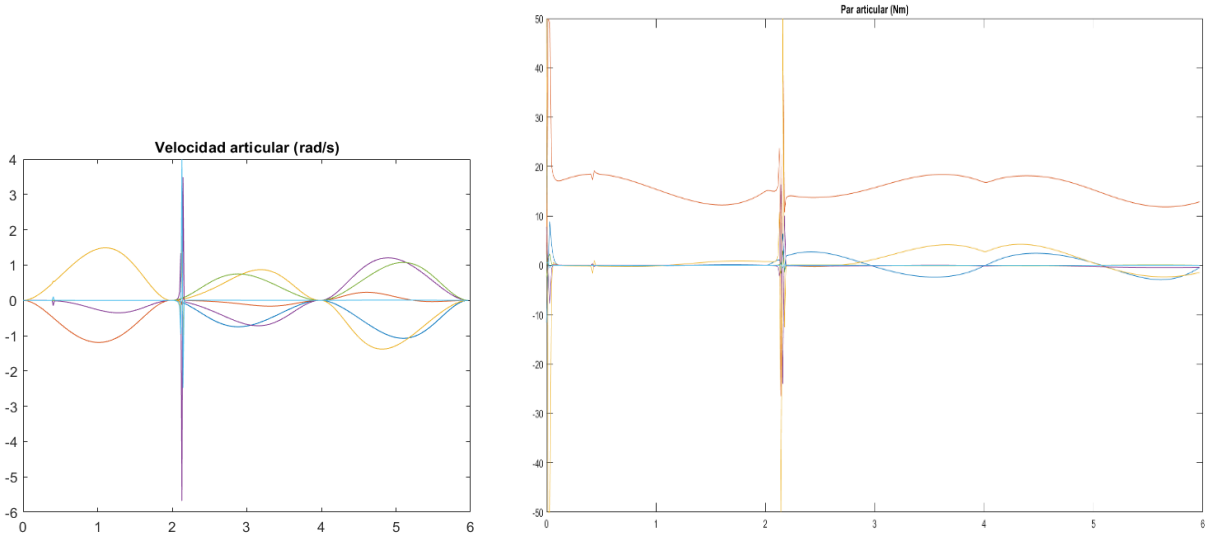


Figura 49: Velocidades y pares articulares controlador Cartesiano simulado

4.3. Resultados Controlador Articular sobre el UR3 real

En este apartado se presentan los resultados obtenidos con el controlador cinemático articular sobre el UR3 real aplicado sobre la base y el codo, es decir, articulaciones 1 y 3.

Se analizará el controlador para ambas articulaciones, de todas las experimentaciones probadas se muestra la comparativa a la simulación con un tiempo de 10 segundos con 2 ganancias distintas cuyas diferencias se mostrarán a continuación.

Para ambos casos la posición de reposo a partir de la cual se generan las trayectorias es la siguiente:

Figura 50: Posición de reposo para la generación de la trayectoria

Partiendo por la base, los resultados obtenidos con una ganancia $Kp = 70$ para un giro de 90 grados en 10 segundos son los mostrados en la Figura 51:

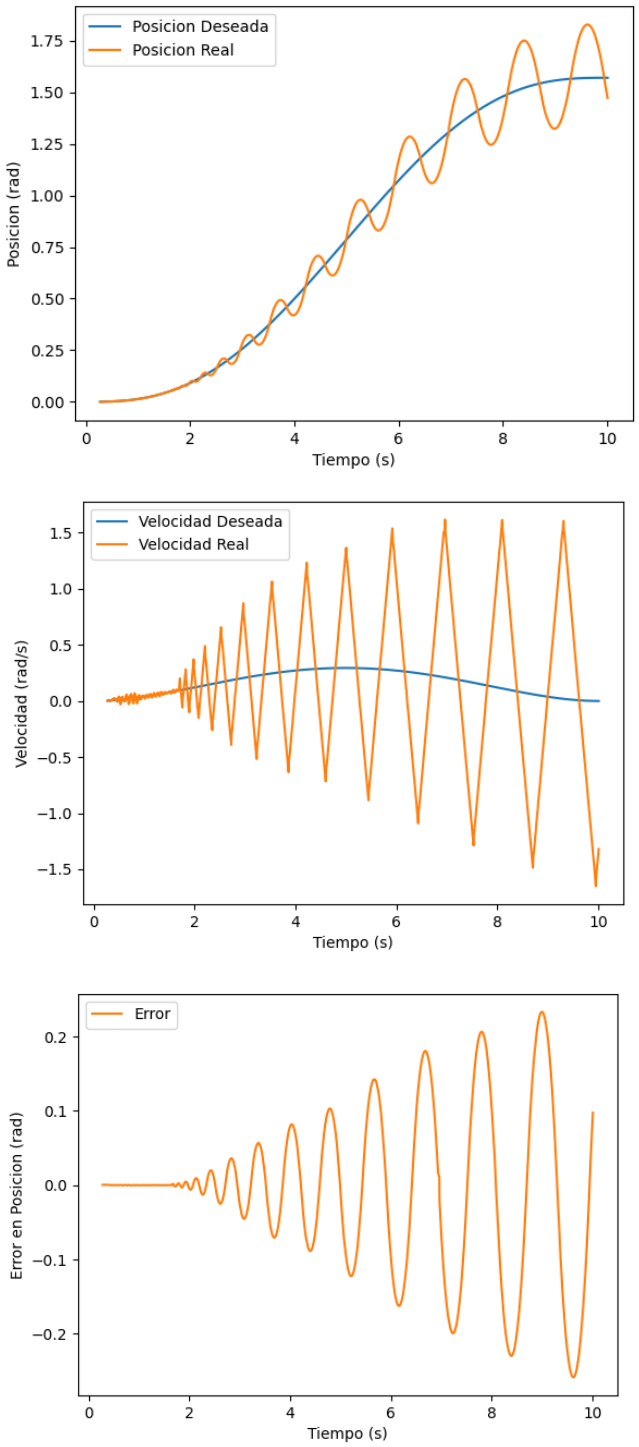
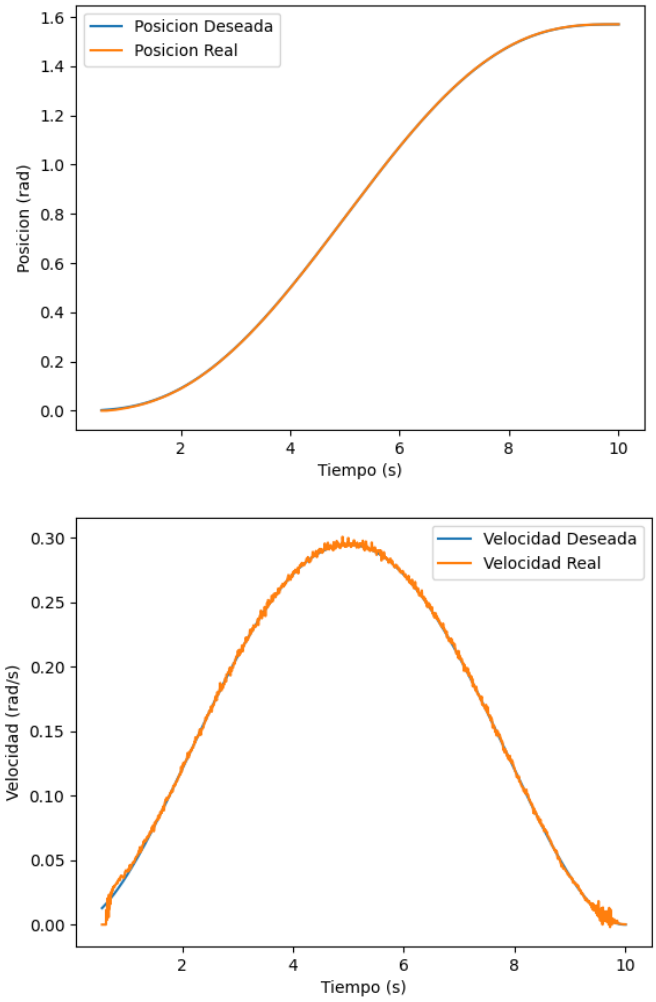


Figura 51: Controlador articular sobre la base con $Kp = 70$

Se puede observar a simple vista que la ganancia es excesivamente alta de forma que produce una respuesta oscilatoria donde si se aumentara la duración de la trayectoria, el error se vería incrementado exponencialmente como se explicó en el apartado 3 de metodología. Por consiguiente, provoca también unos picos en velocidad, nada deseados en un controlador de este tipo.

Sin embargo, para una ganancia *Kp* más baja, de valor 2.5, la respuesta del robot se ve notoriamente mejorada como se muestra en la Figura 52:



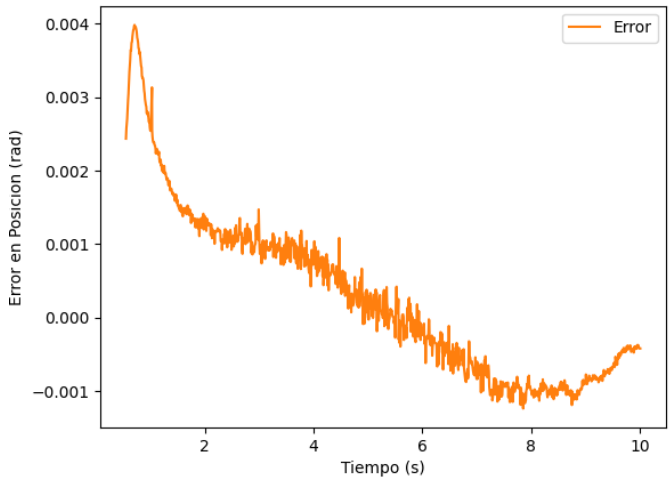


Figura 52: Controlador articular sobre la base con *Kp* = 2.5

A diferencia de la respuesta con la ganancia anterior, se observa un perfecto seguimiento de la referencia por parte del robot en posición. La trayectoria se ejecuta prácticamente con un error insignificante y sin oscilar.

Observando la gráfica de velocidad, también se produce un correcto seguimiento de la referencia, donde aparecen pequeñas interferencias debido a parámetros no modelados en el controlador interno del UR3. Este ruido, a diferencia de la simulación donde no aparece, es una clara prueba de que los gráficos se han obtenido a partir del control aplicado sobre el UR3 real.

Dado que dichas interferencias son insignificantes, si se tuviese que aplicar este controlador para el UR3, de todas las pruebas, las que aportan mejores resultados son para una ganancia *Kp* = 2.5

La posición final del robot tras la ejecución de la trayectoria es la siguiente:

Figura 53: Posición final base tras control articular

Seguidamente, para verificar que el controlador es también válido para cualquier articulación del UR3 se muestran a continuación los resultados de aplicarlo sobre la articulación 3, el codo del robot. La posición inicial de movimiento es la mostrada en la Figura 50. Para este caso, la ganancia a aplicar para conseguir los mejores resultados es $Kp = 1.25$

Luego, los resultados obtenidos de aplicar el controlador articular sobre el codo del UR3 son los mostrados en la Figura 54:

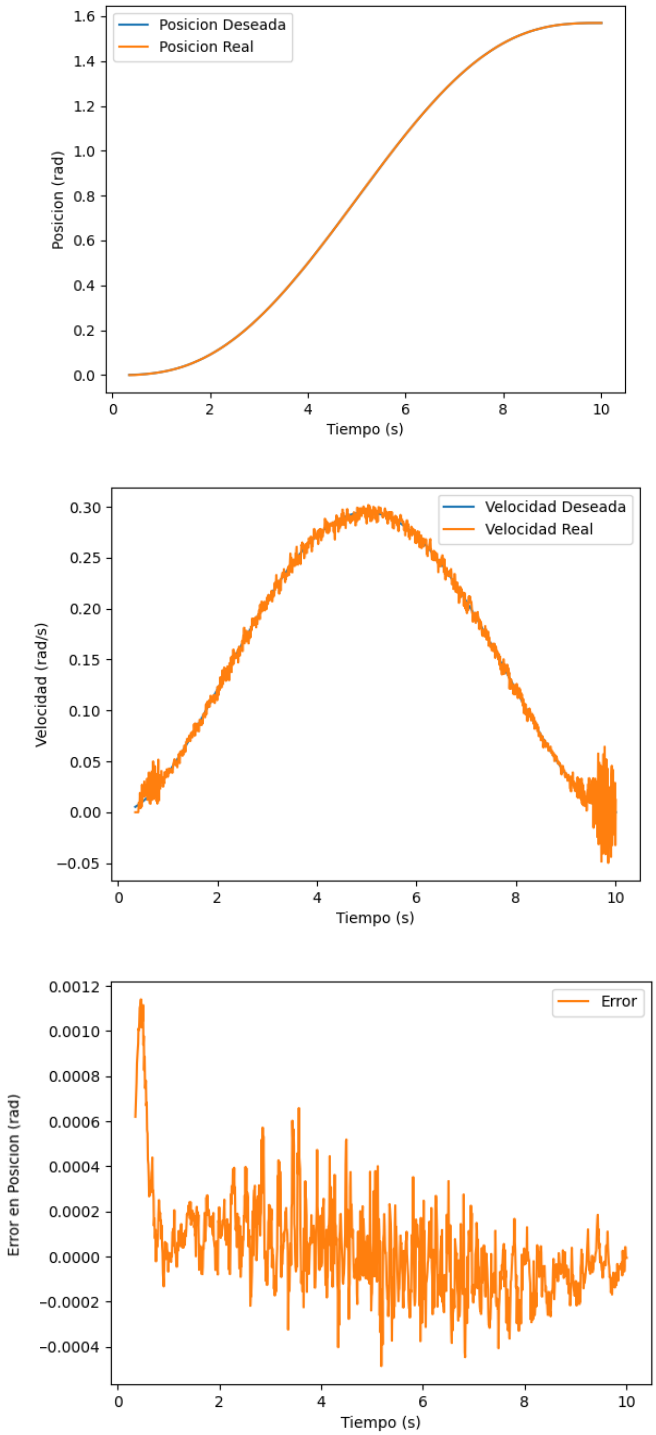


Figura 54: Controlador articular sobre el codo con $Kp = 1.5$

Al igual que para la base con la ganancia adecuada, el controlador responde perfectamente a al seguimiento de la trayectoria generada.

Para este caso en particular, las interferencias en el gráfico de velocidad se ven incrementadas, esto se debe a que el efecto de la gravedad influye mucho más sobre el codo que sobre la base ya que, en este caso, las articulaciones 1 y 3 del UR3 forman 90 grados como se muestra en la Figura 55:

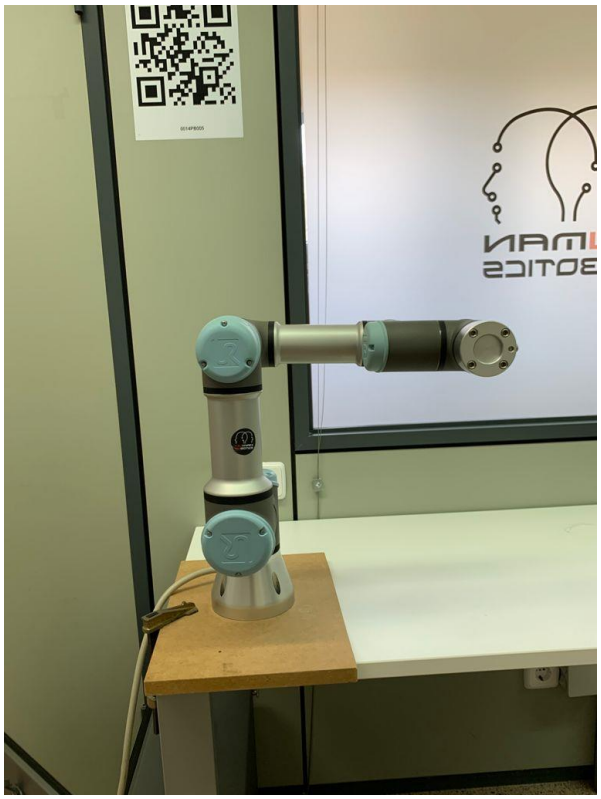


Figura 55: Posición final codo tras el controlador articular

Así, se puede verificar el correcto funcionamiento del controlador articular, tanto a nivel simulado como con el robot real, ya que en ambos casos el robot es capaz de seguir las referencias siempre que la ganancia esté ajustada correctamente.

4.4. Resultados Controlador Cartesiano sobre el UR3 real

En el presente apartado, se presentan los resultados obtenidos con el controlador cinemático Cartesiano para el UR3 real. Se ha implementado desde Python la misma trayectoria que la realizada en la simulación, dividiendo el tiempo total en 3 partes iguales correspondientes a los movimientos en los ejes xyz.

Además, se ha hecho la integración en una trayectoria para que, en vez de ejecutarse en intervalos separados con 2 tiempos distintos, el movimiento del extremo se realicen en un solo eje.

Cabe destacar la influencia de la ganancia en este tipo de controladores, ya que mínimas variaciones del orden de 0.1 unidades, pueden desencadenar en un comportamiento no deseado del robot, a diferencia del articular, donde el rango de ganancias aptas es más amplio.

La necesidad de calcular la matriz Jacobiana a cada iteración puede llegar a causar problemas de tiempo real, ya que el robot envía información por el puerto 30003 cada 8ms. Luego, el PC (maestro) desde donde se ejecuta el controlador, debe ser capaz de recibir, almacenar, acceder, realizar las operaciones pertinentes y hacer el envío de velocidad correspondiente en menos de 8ms, si no, cuando llegue el próximo paquete de datos, el código no estará preparado para poder realizar la lectura correctamente. Esto se soluciona calculando la trayectoria deseada en función del tiempo de lectura, de forma que, aunque cada iteración del programa tiene una duración distinta, el código siempre estará preparado para tomar la lectura que le corresponde en ese instante.

La posición de partida para este controlador para evitar problemas de singularidades es:

q = [0.87,0,-pi/2,-pi/2,0.35,0] como se muestra en la Figura 56:

Figura 56: Posición de partida para el controlador Cartesiano real

Luego, se ha generado una trayectoria de 18 segundos, donde de 0 a 6s, se produce un desplazamiento en z de 0.1m, de 6 a 12s en el eje y, por último, de 12 a 18s en el eje x.

El valor de la ganancia que mejor resultados aporta es Kp = 1.1, cuyos resultados se muestran en las figuras siguientes:

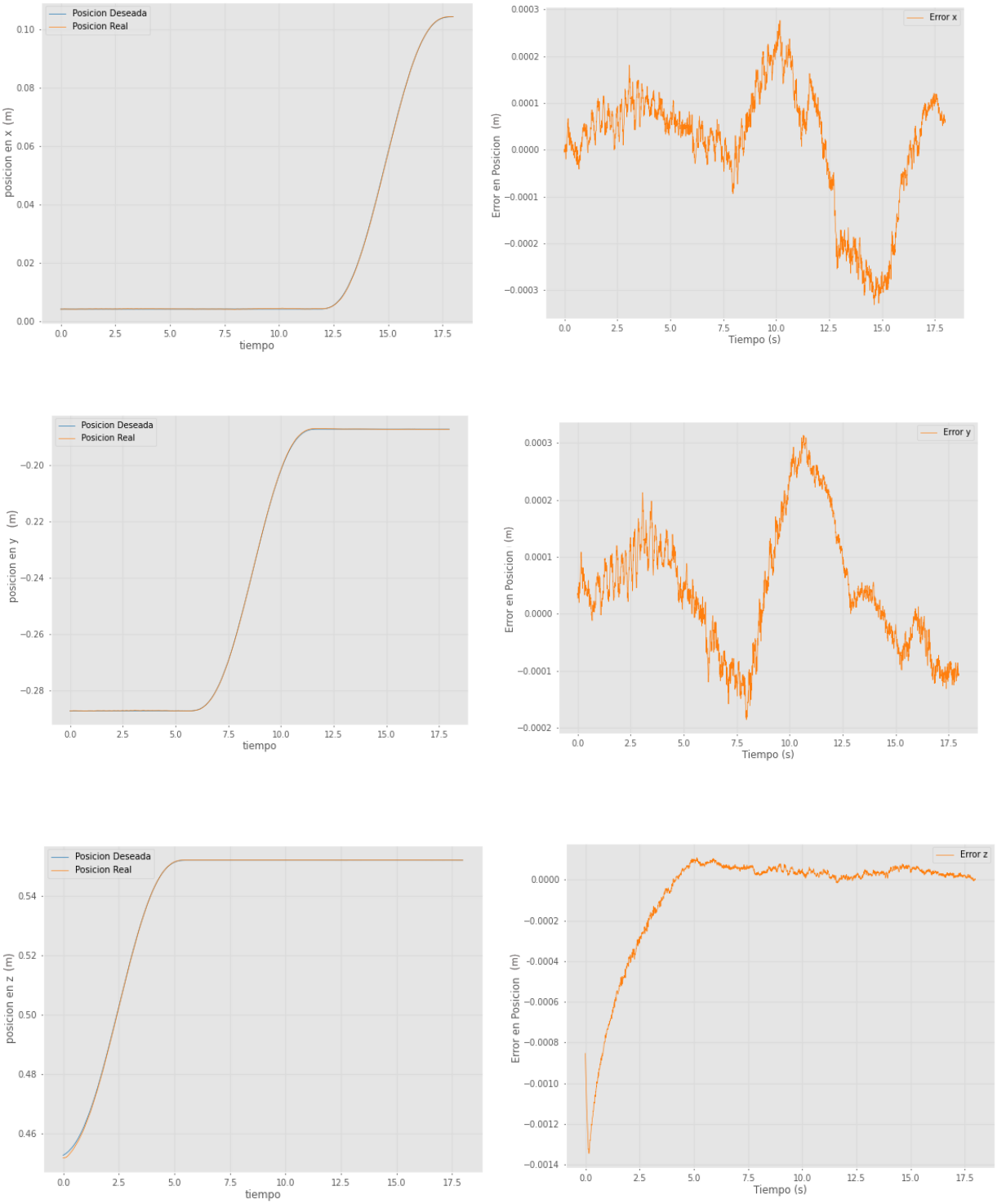


Figura 57: Resultados posición controlador Cartesiano con $Kp = 1.1$ y t = 18s

Como se puede observar en la Figura 57, el seguimiento de las referencias tanto en los 3 ejes para los intervalos de tiempo propuestos se cumple a la perfección. Se puede ver que, en el instante inicial de cada trayectoria, las más desplazada de la posición de partida es en el eje *z*, debido a que el robot está inicialmente parado, una vez terminado el movimiento, el error se mantiene prácticamente nulo igual que en los ejes x e y donde el error medio es aproximadamente es 0.0002 metros, lo que permite afirmar que el comportamiento en posición es totalmente perfecto

En cuanto a la velocidad, el ruido o interferencias que aparecían ya en el controlador articular se ven agravadas, ya que para mover el extremo del robot todas las articulaciones se ven involucradas.

A pesar de estas lecturas con ruido proveniente del robot debido a parámetros físicos no modelados del controlador y lo que se ha comentado anteriormente, si la respuesta en posición es tan buena, debe ser porque los perfiles de velocidad son los adecuados.

En la Figura 58, se muestran las gráficas de velocidad que se traducen en los desplazamientos del robot real mencionados en la Figura 57.

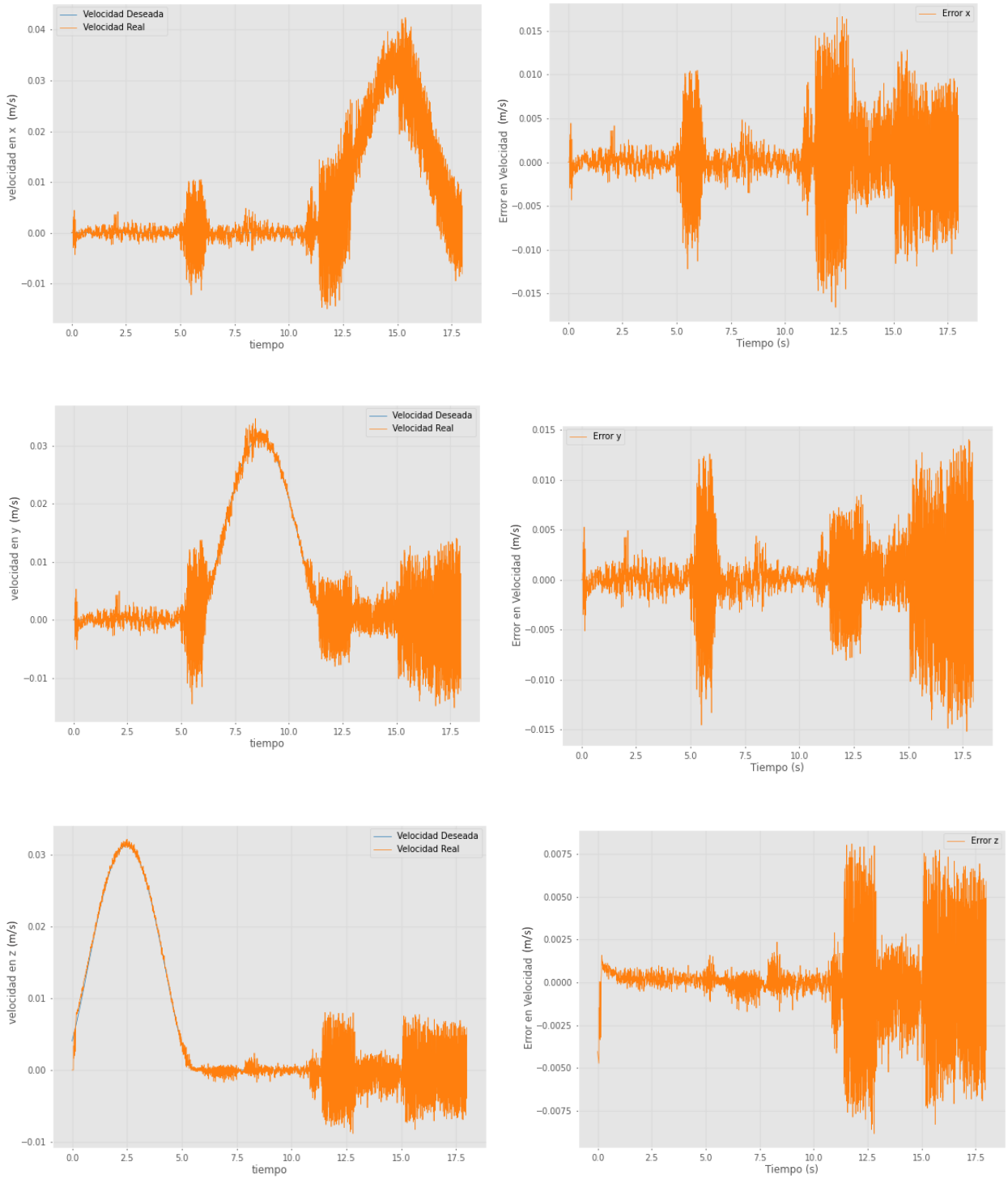


Figura 58: Resultados velocidad controlador Cartesiano con *Kp* = 1.1 y t = 18s

Dichos perfiles representan el comportamiento en posición generado por la trayectoria, ese pequeño ruido hace que el error en x e y sea generalmente mayor que el leído en toda la trayectoria del eje z, pero observando visualmente la ejecución del robot in situ, se garantiza el correcto funcionamiento tanto en el seguimiento de la trayectoria como en fluidez.

La posición final de UR3 tras la ejecución del controlador es la siguiente, desplazada 0.1m en cada eje, a diferencia de la Figura 56 que es la posición inicial.

Figura 59: Posición final del robot tras el controlador Cartesiano

Con el objetivo de verificar cuál es el menor tiempo en el que se puede ejecutar este movimiento se han reducido los tiempos lo máximo posible, de forma que el desplazamiento en z se produce entre 0 y 2 segundos, en el eje y entre 2 y 4 y en x entre 4 y 6 cuyos resultados se muestran a continuación:

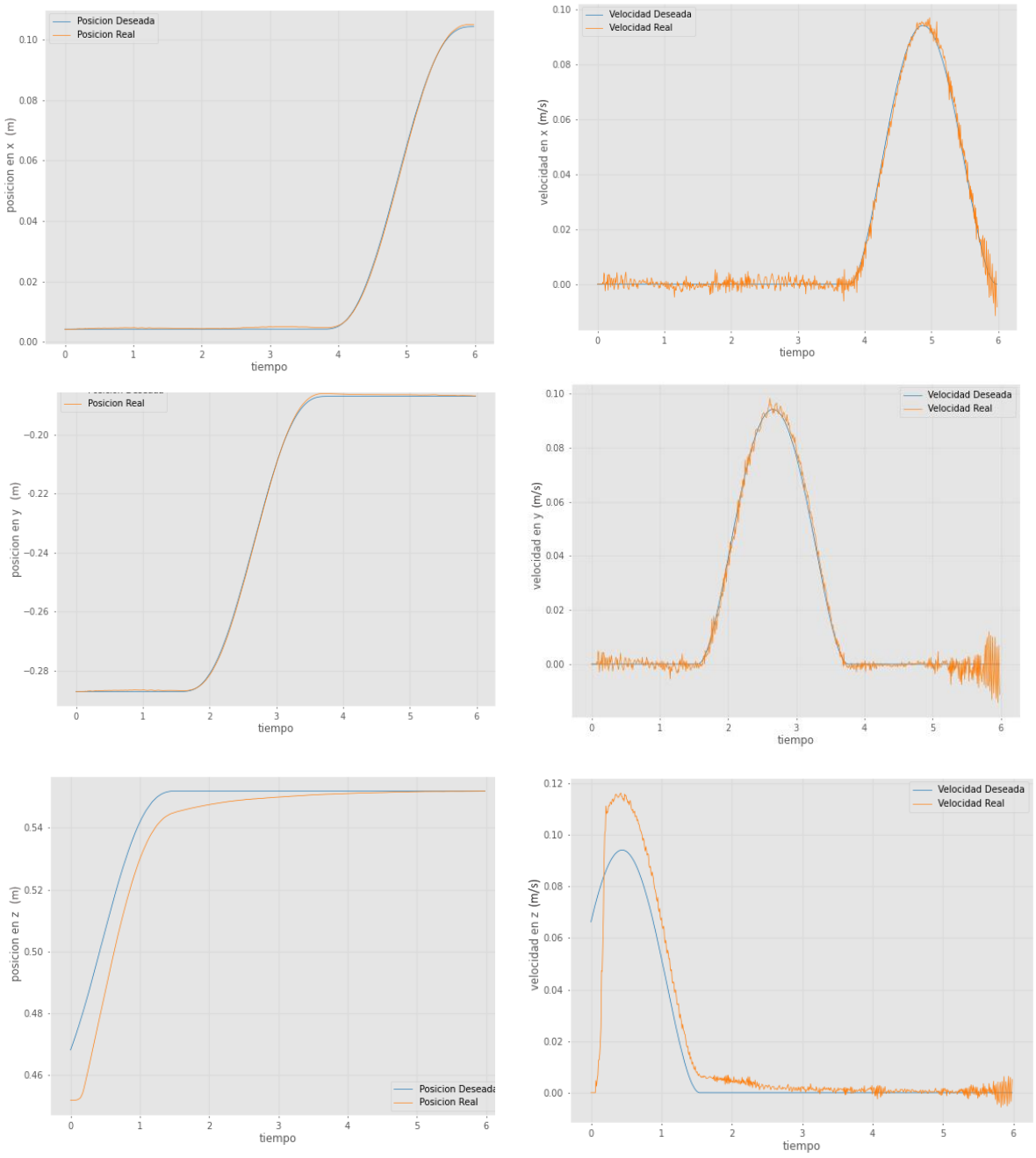


Figura 60: Resultados controlador Cartesiano *Kp* = 0.9 y t = 6s

Se observa que el ruido en las lecturas de velocidad se ve reducido notablemente, además el seguimiento de la referencia es correcto, pero peor con t = 6s que con t = 18s.

El perfil para el eje *z*, no termina de generarse correctamente ya que la primera lectura es la que más tiempo tarda en tomarse e introduce un pequeño retraso que no se puede corregir en tan poco tiempo, a diferencia de como ocurre con los ejes x e y donde el robot ya está en movimiento y los resultados tienen mucho mejor aspecto.

Por último, se muestran los resultados del controlador Cartesiano donde el eje *x* del efector final se desplaza 0.1m de su posición inicial en 4 segundos, donde ahora si que la trayectoria se genera correctamente y se podría afirmar que para el ordenador con el que se ha trabajado, es el tiempo mínimo que se puede fijar para realizar un correcto desplazamiento del extremo.

Para este caso, el seguimiento de la trayectoria es prácticamente perfecto en posición y en velocidad sin aparición de interferencias tan significativas en las lecturas como es el caso del primer resultado del controlador Cartesiano mostrado.

Los resultados se muestran en la Figura 61:

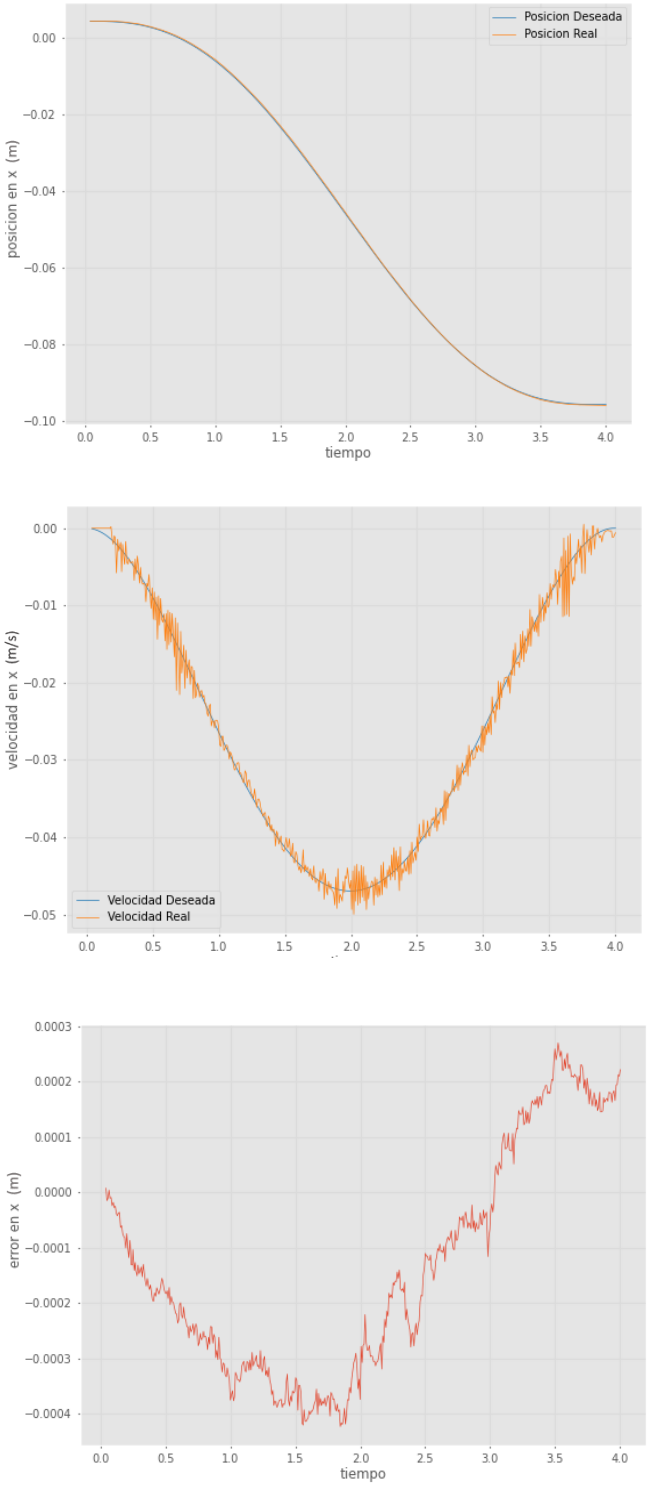


Figura 61: Controlador Cartesiano aplicado sobre el eje x del extremo

Por último, en el siguiente capítulo se exponen las conclusiones del proyecto.

5. Conclusiones

Para finalizar, en este último capítulo una vez expuesto los intereses del problema abordado tanto a nivel teórico como de implementación a nivel simulado y real, se exponen las conclusiones más relevantes obtenidas durante el desarrollo del proyecto.

Primeramente, se ha llevado a cabo una pequeña introducción junto con el estado del arte, de forma que el lector se pueda situar en contexto a partir de los inicios de la robótica y las estrategias de control cinemático que se han implementado durante el desarrollo del proyecto.

Se ha abordado la metodología que se ha seguido para desarrollar el presente TFM donde se ha expuesto los procedimientos necesarios a nivel de simulación y en la realidad para el UR3 de *Universal Robots.*

Las principales diferencias a destacar entre implementar el controlador simulado y en la realidad son principalmente el problema de la sincronización y la forma de tomar las lecturas. Ya que la trayectoria que se genera en *MATLAB* tiene un periodo de muestreo fijo, si se establecen los pasos del simulador al mismo valor que dicho tiempo es imposible que se desincronicen los programas ya que los envíos por el *socket* se hacen a través de una señal de disparo que lo evita. De este modo se ha podido corroborar que es posible implementar controladores cinemáticos a nivel articular y Cartesiano para el UR3, ya que además de posiciones articulares, permite el envío de muchos más parámetros a partir de los cuales se pueden diseñar este tipo de controladores.

Cuando se trabaja con el robot real, el puerto 30003 de comunicaciones en tiempo real envía un paquete de 1140 *bytes* con toda la información del robot cada 8 milisegundos, de forma que no se puede reclamar el parámetro que se desee de primeras, sino que hay que recibir el paquete completo, y acceder a las posiciones de memoria donde se encuentran los datos necesarios en el orden que especifica el fabricante. Luego una vez conseguido la información debe emplearse para hacer los cálculos cinemáticos correspondientes y hacer el envío de velocidad. Si este tiempo es mayor de 8ms, el código no estará preparado para recibir el paquete del UR3 por lo que es muy complicado realizar un control de este tipo con un portátil estándar.

La solución aplicada, fue calcular la trayectoria deseada en función del tiempo de lectura del *socket*, de forma que cada envío del robot corresponderá con la iteración del bucle en la que se encuentren.

Este problema surgió a raíz de que resultaba imposible tomar lecturas desde MATLAB a tiempo real como se ha expuesto en el apartado de metodología, luego, se tuvo que emplear Python como alternativa para conseguir unas lecturas mucho más veloces que permitiesen implementar un controlador de este tipo.

En cuanto a los controladores articulares y Cartesianos, se puede confirmar la correcta implementación tanto en simulación como con el UR3 real, ya que se han podido generar las trayectorias correctamente y gracias al diseño de los controladores el robot es capaz de seguir las referencias tanto en posición como en velocidad adecuadamente.

Una posible mejora a bajo nivel sería implementar el controlador en C++, de forma que, al ser un lenguaje compilado, el tiempo de ejecución de las lecturas sería incluso más bajo que en Python. Además, si se implementase un *thread* (hilo), donde solo se tomase la lectura y se pudiese acceder a ella desde el bucle de controlador, es probable que disminuyesen los problemas de tiempo real, principal problema en las comunicaciones de este proyecto, lo que se tendrá en cuenta para futuras investigaciones.

Por último, se corrobora y verifica la correcta implementación de los controladores cinemáticos articulares y Cartesianos para el robot UR3 de *Universal Robots* cumpliendo los objetivos tanto a nivel de simulación como en la realidad como se ha expuesto a lo largo del documento.

6. Lista de acrónimos y abreviaturas

API ..... Application Programming Interface

DH ..... Denavit - Hartenberg

GDL ..... Grados de Libertad

IDE ..... Integrated Development Environment

MATLAB ..... Matrix Laboratory

TCP/IP ..... Transmission Control Protocol/Internet Protocol

TFM ..... Trabajo Final de Master

UR ..... Universal Robots

7. Bibliografía

1. Čapek, K. (1920/2017). *Robots Universal Rossum*. Libros Mablaz.
2. Asimov, I. (1950/2008). *I Robot.* Oxford University Press.
3. Gómez, G. J. (2020). *Cinemática y Dinámica de los Sistemas Robóticos.* UA.
4. Gómez, G. J. (2020). *Control cinemático de robots.* UA.
5. https://www.universal-robots.com/es/
6. https://www.universal-robots.com/articles/ur/interface-communication/remote-control-via-tcpip/
7. https://www.universal-robots.com/es/productos/
8. https://www.coppeliarobotics.com/
9. https://es.mathworks.com/products/matlab.html
10. https://github.com/petercorke/robotics-toolbox-matlab
11. https://github.com/petercorke/robotics-toolbox-matlab/blob/master/tpoly.m
12. https://forum.universal-robots.com/t/getting-robot-position-using-rtde-and-python-3-x/3411
13. https://www.zacobria.com/universal-robots-zacobria-forum-hints-tips-how-to/script-via-socket-connection
14. https://petercorke.github.io/robotics-toolbox-python/intro.html#robot-models
15. https://petercorke.github.io/robotics-toolbox-python/intro.html#trajectories
16. F. Torres, J. Pomares, P. Gil, S. Puente, R. Aracil. Prentice Hall. (2002). *Robots y Sistemas Sensoriales.*
17. John Craig. Addison Wesley. (2004). ISBN: 978-0133489798. *Introduction to Robotics: Mechanics and Control.*
18. A. Barrientos, L. F. Peñín, C. Balaguer, R. Aracil. Mc Graw Hill. (2007). ISBN: 978-84-481-5636-7. *Fundamentos de Robótica.*
19. Siciliano, B., Sciavicco, L., Villani, L., Oriolo, G. Springer-Verlag, London (2010). ISBN: 978-1-84628-641-4. *Robotics: Modelling, Planning and Control.*

20. Siciliano, B., Khatib, O., Springer-Verlag, Berlin (2008) ISBN: ISBN: 978-3-540-23957-4. *Handbook of Robotics.*

21. Lewis, F. L., Dawson, D. M., Abdallah, C. T, CRC Press, New York (2003) ISBN: 978-0824740726. *Robot Manipulator Control: Theory and Practice.*

Se puede acceder a los códigos implementados en el presente TFM desde el siguiente enlace:

https://drive.google.com/open?id=1HmQVgAyRZ7q9ApjhZzb4MGZ7M3xyan0D

Códigos

En este apartado se muestran los códigos implementados para los controladores cinemáticos articulares y Cartesianos simulados y reales sobre un robot UR3.

Código 1: Controlador Cinemático Articular Simulado

```
% simRemoteApi.start(23000,1300,false,true);

clear;clc;close all;

disp('Inicio del programa');
vrep=remApi('remoteApi');
vrep.simxFinish(-1); % Por si hubiera una conexión anterior abierta
clientID=vrep.simxStart('127.0.0.1',23000,true,true,5000,0.01);

if (clientID>-1)
    disp('Conectado al servidor remoto de la API');

    % Seleccionamos la primera articulación del robot UR
    [~,ur3]=vrep.simxGetObjectHandle(clientID,'UR3_joint1',vrep.simx_opmode_oneshot_wait);

    % Obtenemos posición articular
    [~,q0] =vrep.simxGetJointPosition(clientID, ur3, vrep.simx_opmode_oneshot_wait);

    % Inicialización de la ganancia K

    Kp = 20;

    % Cálculo de la trayectoria y su velocidad
    qf = q0+deg2rad(89);
    t_muestreo = 0.005;
    t = 0:t_muestreo:2;

    [Q_deseada,Qd_deseada]=tpoly(q0,qf,t);

    % Inicialiazaciones (si se requieren)
    ep = zeros(1,length(Q_deseada));
    torque = zeros(1,length(Q_deseada));
    q_real = zeros(1,length(Q_deseada));
    qd_real = zeros(1,length(Q_deseada));


    % Bucle de control

    %Habilita la sincronización y comienza la simulacion
    vrep.simxStartSimulation(clientID,vrep.simx_opmode_oneshot);
    vrep.simxSynchronous(clientID,1);
    for i=1:length(Q_deseada)

        %Leer q de vrep
        [~,qd_real(i)] = vrep.simxGetObjectFloatParameter(clientID, ur3,2012,vrep.simx_opmode_oneshot_wait);
        [~,q_real(i)] = vrep.simxGetJointPosition(clientID, ur3, vrep.simx_opmode_oneshot_wait);
        [~,torque(i)] = vrep.simxGetJointForce(clientID, ur3, vrep.simx_opmode_oneshot_wait);
```

```
49
50 -            ep(i) = Q_deseada(i)-q_real(i);
51 -            qd_PID = ep(i)*Kp;
52 -            qd_robot = qd_PID + Qd_deseada(i);
53
54              %Mandar qd_robot a vrep
55 -            vrep.simxSetJointTargetVelocity(clientID,ur3,qd_robot,vrep.simx_opmode_oneshot);
56              %Enviar de forma sincrona los ultimos valores
57 -            vrep.simxSynchronousTrigger(clientID);
58
59 -        end
60
61 -        vrep.simxPauseSimulation(clientID,vrep.simx_opmode_oneshot);
62
63
64 -        t_plot = 0:t_muestreo:2;
65
66 -        figure;
67 -        subplot(221);
68 -        plot(t_plot,q_real);
69 -        hold on
70 -        plot(t_plot,Q_deseada);
71 -        title('Posicion (rad)');
72 -        legend('real','deseada');
73 -        grid;
74 -        subplot(222);
75 -        plot(t_plot,ep);
76 -        title('Error Posicion (rad)');
77 -        grid;
78 -        subplot(223);
79 -        plot(t_plot,qd_real);
80 -        hold on
81 -        plot(t_plot,Qd_deseada);
82 -        title('Velocidad (rad/s)');
83 -        legend('real','deseada');
84 -        grid;
85 -        subplot(224);
86 -        plot(t_plot,torque);
87 -        title('Par (Nm)');
88 -        grid;
89
90          % Antes de cerrar la conexión con V-REP, aseguramos que el último
91          % comando enviado tuvo tiempo de llegar
92 -        vrep.simxGetPingTime(clientID);
93
94          % Ahora se cierra la conexión con V-REP:
95 -        vrep.simxFinish(clientID);
96 -    else
97 -        disp('Fallo en el intento de conexión al servidor remoto de la API');
98 -    end
99 -    vrep.delete(); % Llamada al destructor
100
101 -   disp('Fin del programa');
```

Código 2: Controlador Cinemático Cartesiano simulado

```
% simRemoteApi.start(23000,1300,false,true);

mdl_ur3;
ur3.offset = [-pi/2,-pi/2,0,-pi/2,0,-pi/2];

disp('Program started');
vrep=remApi('remoteApi'); % Usando el archivo prototipo (remoteApiProto.m)
vrep.simxFinish(-1); % En caso de que hubiese conexiones abiertas, para cerrarlas
clientID=vrep.simxStart('127.0.0.1',23000,true,true,5000,5);


if (clientID>-1)
    disp('Connected to remote API server');

    % Recibimos datos de 'prueba'
    [res,objs]=vrep.simxGetObjects(clientID,vrep.sim_handle_all,vrep.simx_opmode_blocking);

    % Recibimos datos del UR3

    [~,ur1]=vrep.simxGetObjectHandle(clientID,'UR3_joint1',vrep.simx_opmode_oneshot_wait);
    [~,ur2]=vrep.simxGetObjectHandle(clientID,'UR3_joint2',vrep.simx_opmode_oneshot_wait);
    [~,ur_3]=vrep.simxGetObjectHandle(clientID,'UR3_joint3',vrep.simx_opmode_oneshot_wait);
    [~,ur4]=vrep.simxGetObjectHandle(clientID,'UR3_joint4',vrep.simx_opmode_oneshot_wait);
    [~,ur5]=vrep.simxGetObjectHandle(clientID,'UR3_joint5',vrep.simx_opmode_oneshot_wait);
    [~,ur6]=vrep.simxGetObjectHandle(clientID,'UR3_joint6',vrep.simx_opmode_oneshot_wait);
    [~,ur_tool]=vrep.simxGetObjectHandle(clientID,'UR3_connection',vrep.simx_opmode_oneshot_wait);
    [~,ur_base]=vrep.simxGetObjectHandle(clientID,'UR3',vrep.simx_opmode_oneshot_wait);

    [~,pos] =vrep.simxGetObjectPosition(clientID, ur_tool, ur_base, vrep.simx_opmode_oneshot_wait);
    [~,ori] =vrep.simxGetObjectOrientation(clientID, ur_tool, ur_base, vrep.simx_opmode_oneshot_wait);

    vectorTiempo = 0:0.005:0.66;
    [s1X,s1dX]=tpoly(pos(1),pos(1),vectorTiempo,0,0);
    [s1Y,s1dY]=tpoly(pos(2),pos(2),vectorTiempo,0,0);
    [s1Z,s1dZ]=tpoly(pos(3),pos(3)+0.1,vectorTiempo,0,0);

    [s2X,s2dX]=tpoly(pos(1),pos(1),vectorTiempo,0,0);
    [s2Y,s2dY]=tpoly(pos(2),pos(2)+0.1,vectorTiempo,0,0);
    [s2Z,s2dZ]=tpoly(pos(3)+0.1,pos(3)+0.1,vectorTiempo,0,0);

    [s3X,s3dX]=tpoly(pos(1),pos(1)+0.1,vectorTiempo,0,0);
    [s3Y,s3dY]=tpoly(pos(2)+0.1,pos(2)+0.1,vectorTiempo,0,0);
    [s3Z,s3dZ]=tpoly(pos(3)+0.1,pos(3)+0.1,vectorTiempo,0,0);


    sX = [s1X;s2X;s3X];
    sY = [s1Y;s2Y;s3Y];
    sZ = [s1Z;s2Z;s3Z];
    sdX = [s1dX;s2dX;s3dX];
    sdY = [s1dY;s2dY;s3dY];
    sdZ = [s1dZ;s2dZ;s3dZ];

```

```
54          % Bucle de control
55          % Ganancia
56 -        K=0.03; %(50,90,-90,0,0,0)°
57
58          % Habilita la sincronización y comienza la simulacion
59 -        vrep.simxStartSimulation(clientID,vrep.simx_opmode_oneshot_wait);
60 -        vrep.simxSynchronous(clientID,1);
61 -        vrep.simxSynchronousTrigger(clientID);
62
63 -        for i=1:max(size(sdZ))
64 -            [~,pos] =vrep.simxGetObjectPosition(clientID, ur_tool, ur_base, vrep.simx_opmode_oneshot_wait);
65 -            [~,ori] =vrep.simxGetObjectOrientation(clientID, ur_tool, ur_base, vrep.simx_opmode_oneshot_wait);
66 -            [~,q1] =vrep.simxGetJointPosition(clientID, ur1, vrep.simx_opmode_oneshot_wait);
67 -            [~,q2] =vrep.simxGetJointPosition(clientID, ur2, vrep.simx_opmode_oneshot_wait);
68 -            [~,q3] =vrep.simxGetJointPosition(clientID, ur_3, vrep.simx_opmode_oneshot_wait);
69 -            [~,q4] =vrep.simxGetJointPosition(clientID, ur4, vrep.simx_opmode_oneshot_wait);
70 -            [~,q5] =vrep.simxGetJointPosition(clientID, ur5, vrep.simx_opmode_oneshot_wait);
71 -            [~,q6] =vrep.simxGetJointPosition(clientID, ur6, vrep.simx_opmode_oneshot_wait);
72 -            [~,tau(1,i)] =vrep.simxGetJointForce(clientID, ur1, vrep.simx_opmode_oneshot_wait);
73 -            [~,tau(2,i)] =vrep.simxGetJointForce(clientID, ur2, vrep.simx_opmode_oneshot_wait);
74 -            [~,tau(3,i)] =vrep.simxGetJointForce(clientID, ur_3, vrep.simx_opmode_oneshot_wait);
75 -            [~,tau(4,i)] =vrep.simxGetJointForce(clientID, ur4, vrep.simx_opmode_oneshot_wait);
76 -            [~,tau(5,i)] =vrep.simxGetJointForce(clientID, ur5, vrep.simx_opmode_oneshot_wait);
77 -            [~,tau(6,i)] =vrep.simxGetJointForce(clientID, ur6, vrep.simx_opmode_oneshot_wait);
78 -            [~,vel_lineal] = vrep.simxGetObjectVelocity(clientID, ur_tool,vrep.simx_opmode_oneshot_wait);
79
80 -            vel_real(1,i)=vel_lineal(1);
81 -            vel_real(2,i)=vel_lineal(2);
82 -            vel_real(3,i)=vel_lineal(3);
83 -            realposX(i)=pos(1);
84 -            realposY(i)=pos(2);
85 -            realposZ(i)=pos(3);
86 -            pdotX=sdX(i)+K*(sX(i)-pos(1));
87 -            pdotY=sdY(i)+K*(sY(i)-pos(2));
88 -            pdotZ=sdZ(i)+K*(sZ(i)-pos(3));
89 -            q=[double(q1) double(q2) double(q3) double(q4) double(q5) double(q6)];
90 -            jacobiana = ur3.jacob0(q);
91
92
93 -            qdot=inv(jacobiana)*[pdotX pdotY pdotZ 0 0 0]';
94
95 -            realvel(:,i)=qdot;
96
97 -            returnCode=vrep.simxSetJointTargetVelocity(clientID,ur1,qdot(1),vrep.simx_opmode_oneshot);
98 -            returnCode=vrep.simxSetJointTargetVelocity(clientID,ur2,qdot(2),vrep.simx_opmode_oneshot);
99 -            returnCode=vrep.simxSetJointTargetVelocity(clientID,ur_3,qdot(3),vrep.simx_opmode_oneshot);
100 -           returnCode=vrep.simxSetJointTargetVelocity(clientID,ur4,qdot(4),vrep.simx_opmode_oneshot);
101 -           returnCode=vrep.simxSetJointTargetVelocity(clientID,ur5,qdot(5),vrep.simx_opmode_oneshot);
102 -           returnCode=vrep.simxSetJointTargetVelocity(clientID,ur6,qdot(6),vrep.simx_opmode_oneshot);
103         %   % Contador termina
104 -           vrep.simxSynchronousTrigger(clientID);
105         %   % Contador empieza
106 -       end
```

```
107
108 -       vrep.simxPauseSimulation(clientID,vrep.simx_opmode_oneshot);
109
110 -       vectorTiempoPlot = 0:0.015:6-0.03;
111
112         % Ploteado de las figuras con la posición real del robot, el error
113         % en posición, la velocidad real, el error en velocidad y el par real.
114
115 -       figure;
116 -       title('Velocidad Cartesiana');
117 -       subplot(231)
118 -       plot(vectorTiempoPlot,sdX);
119 -       hold on
120 -       plot(vectorTiempoPlot,vel_real(1,:));
121 -       title('Eje X (m/s)');
122 -       subplot(232)
123 -       plot(vectorTiempoPlot,sdY);
124 -       hold on
125 -       plot(vectorTiempoPlot,vel_real(2,:));
126 -       title('Eje Y (m/s)');
127 -       subplot(233)
128 -       plot(vectorTiempoPlot,sdZ);
129 -       hold on
130 -       plot(vectorTiempoPlot,vel_real(3,:));
131 -       title('Eje Z (m/s)');
132
133 -       subplot(234)
134 -       plot(vectorTiempoPlot,sdX-vel_real(1,:)');
135 -       title('Error eje X (m/s)')
136 -       subplot(235)
137 -       plot(vectorTiempoPlot,sdY-vel_real(2,:)');
138 -       title('Error eje Y (m/s)')
139 -       subplot(236)
140 -       plot(vectorTiempoPlot,sdZ-vel_real(3,:)');
141 -       title('Error eje Z (m/s)')
142
143 -       figure;
144 -       plot(vectorTiempoPlot,realvel);
145 -       title('Velocidad articular (rad/s)')
146
147 -       figure;
148 -       plot(vectorTiempoPlot,tau);
149 -       title('Par articular (Nm)');
150
151 -       figure;
152 -       title('Posicion Cartesiana');
153 -       subplot(231)
154 -       plot(vectorTiempoPlot,sX);
155 -       hold on
156 -       plot(vectorTiempoPlot,realposX);
157 -       title('Eje X (m)');
158 -       subplot(232)
159 -       plot(vectorTiempoPlot,sY);
```

```
160 -         hold on
161 -         plot(vectorTiempoPlot,realposY);
162 -         title('Eje Y (m)');
163 -         subplot(233)
164 -         plot(vectorTiempoPlot,sZ);
165 -         hold on
166 -         plot(vectorTiempoPlot,realposZ);
167 -         title('Eje Z (m)');
168 -         subplot(234)
169 -         plot(vectorTiempoPlot,sX-realposX');
170 -         title('Error eje X (m)')
171 -         subplot(235)
172 -         plot(vectorTiempoPlot,sY-realposY');
173 -         title('Error eje Y (m)')
174 -         subplot(236)
175 -         plot(vectorTiempoPlot,sZ-realposZ');
176 -         title('Error eje Z (m)')
177
178           % Confirmamos último envio y Cerramos V-REP
179 -         vrep.simxGetPingTime(clientID);
180 -         vrep.simxFinish(clientID);
181 -     else
182 -         disp('Failed connecting to remote API server');
183 -     end
184 -     vrep.delete(); % Llamada destructor!
185 -     disp('Program ended');
```

Código 3: Controlador Cinemático Articular robot real

```
import socket
import time
import struct
import numpy as np
from matplotlib import pyplot as plt
from tpoly import tpoly

HOST = '192.168.2.2'
PORT_30003 = 30003
t = np.arange(0,2,0.008)

p, pd, coeffs_1, coeffs_d = tpoly(0,1.57,t,0,0)

print ("Starting Program")

qf = 1.57
q_real = []
pos = []
vel = []
error = []
qd_real = []
save_duration = []
qd_zeros = [0]*6
ep = []
Kp = 2.5
i = 0
flag = 1
start = time.time()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT_30003))

while(flag):

        packet_1 = s.recv(4)
        packet_2 = s.recv(8)
        packet_3 = s.recv(48)
        packet_4 = s.recv(48)
        packet_5 = s.recv(48)
        packet_6 = s.recv(48)
        packet_7 = s.recv(48)
        packet_8_1 = s.recv(8)
        packet_8_2 = s.recv(8)
        packet_8_3 = s.recv(8)
```

```
packet_8_4 = s.recv(8)
packet_8_5 = s.recv(8)
packet_8_6 = s.recv(8)
packet_9_1 = s.recv(8)
packet_9_2 = s.recv(8)
packet_9_3 = s.recv(8)
packet_9_4 = s.recv(8)
packet_9_5 = s.recv(8)
packet_9_6 = s.recv(8)
packets_extra = s.recv(792)

q1 = str(packet_8_1)
q1 = struct.unpack("!d", packet_8_1)[0]
q2 = str(packet_8_2)
q2 = struct.unpack("!d", packet_8_2)[0]
q3 = str(packet_8_3)
q3 = struct.unpack("!d", packet_8_3)[0]
q4 = str(packet_8_4)
q4 = struct.unpack("!d", packet_8_4)[0]
q5 = str(packet_8_5)
q5 = struct.unpack("!d", packet_8_5)[0]
q6 = str(packet_8_6)
q6 = struct.unpack("!d", packet_8_6)[0]
q = [q1, q2, q3, q4, q5, q6]
print ("q = ", q)
q_real.append(q1)

qd1 = str(packet_9_1)
qd1 = struct.unpack("!d", packet_9_1)[0]
qd2 = str(packet_9_2)
qd2 = struct.unpack("!d", packet_9_2)[0]
qd3 = str(packet_9_3)
qd3 = struct.unpack("!d", packet_9_3)[0]
qd4 = str(packet_9_4)
qd4 = struct.unpack("!d", packet_9_4)[0]
qd5 = str(packet_9_5)
qd5 = struct.unpack("!d", packet_9_5)[0]
qd6 = str(packet_9_6)
qd6 = struct.unpack("!d", packet_9_6)[0]
qd = [qd1, qd2, qd3, qd4, qd5, qd6]
print ("qd = ", qd)
qd_real.append(qd1)


end = time.time()
duration = end - start
save_duration.append(duration)
```

```
        if duration < 2:
            posicion = np.polyval(coeffs_1, duration)
            velocidad= np.polyval(coeffs_d, duration)
        else:
            posicion = qf
            velocidad = 0
            flag = 0

        pos.append(posicion)
        vel.append(velocidad)


        ep = posicion - q_real[i]
        qd_PID = ep*Kp
        qd_robot = qd_PID + velocidad
        error.append(ep)

        string = 'speedj([%f, 0, 0, 0, 0, 0], 5, 2)' % (qd_robot) + '\n'
        strcode = string.encode()
        s.send (strcode)

        i = i + 1
        print ("Program finish")

s.close()
plt.figure(1)
plt.plot(save_duration, pos, 'tab:blue', label='Posicion Deseada')
plt.plot(save_duration, q_real,'tab:orange', label='Posicion Real')
plt.legend()
plt.xlabel('Tiempo (s)')
plt.ylabel('Posicion (rad)')
plt.figure(2)
plt.plot(save_duration, vel, 'tab:blue', label='Velocidad Deseada')
plt.plot(save_duration, qd_real, 'tab:orange', label='Velocidad Real')
plt.legend()
plt.xlabel('Tiempo (s)')
plt.ylabel('Velocidad (rad/s)')
plt.figure(3)
plt.plot(save_duration, error, 'tab:orange', label='Error')
plt.legend()
plt.xlabel('Tiempo (s)')
plt.ylabel('Error en Posicion (rad)')

plt.show()
```

Código 4: Controlador Cinemático Cartesiano robot real

```
import socket
import time
import struct
import numpy as np
from matplotlib import pyplot as plt
from tpoly import tpoly
import roboticstoolbox as rtb


save_duration = []
posxx = []
velxx = []
posyy = []
velyy = []
poszz = []
velzz = []
K = 1
i = 0
flag = 1
qf = 0.1
pos_X = []
vel_X =[]
pos_Y = []
vel_Y =[]
pos_Z =[]
vel_Z = []
robot = rtb.models.DH.UR3()

xtcp = 0.00424
ytcp = -0.28712
ztcp = 0.45192


HOST = '192.168.2.2' # The remote host
PORT_30003 = 30003

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    #s.settimeout(1)
s.connect((HOST, PORT_30003))

packet_1 = s.recv(4)
packet_2 = s.recv(8)
packet_3 = s.recv(48)
packet_4 = s.recv(48)
```

```
packet_5 = s.recv(48)
packet_6 = s.recv(48)
packet_7 = s.recv(48)
packet_8_1 = s.recv(8)
packet_8_2 = s.recv(8)
packet_8_3 = s.recv(8)
packet_8_4 = s.recv(8)
packet_8_5 = s.recv(8)
packet_8_6 = s.recv(8)
packet_9 = s.recv(48)
packet_10 = s.recv(48)
packet_11 = s.recv(48)
packet_12_1 = s.recv(8)
packet_12_2 = s.recv(8)
packet_12_3 = s.recv(8)
packet_12_456 = s.recv(24)
#packet_12 = s.recv(48)
packet_13_1 = s.recv(8)
packet_13_2 = s.recv(8)
packet_13_3 = s.recv(8)
packet_13_456 = s.recv(24)
packets_extra = s.recv(600)
s.close()


# POSICIONES ARTICULARES
q1 = str(packet_8_1)
q1 = struct.unpack("!d", packet_8_1)[0]
q2 = str(packet_8_2)
q2 = struct.unpack("!d", packet_8_2)[0]
q3 = str(packet_8_3)
q3 = struct.unpack("!d", packet_8_3)[0]
q4 = str(packet_8_4)
q4 = struct.unpack("!d", packet_8_4)[0]
q5 = str(packet_8_5)
q5 = struct.unpack("!d", packet_8_5)[0]
q6 = str(packet_8_6)
q6 = struct.unpack("!d", packet_8_6)[0]

# POSICION TCP
pos_x = str(packet_12_1)
pos_x = struct.unpack("!d", packet_12_1)[0]
pos_y = str(packet_12_2)
pos_y = struct.unpack("!d", packet_12_2)[0]
pos_z = str(packet_12_3)
pos_z = struct.unpack("!d", packet_12_3)[0]
```

```
t = np.arange(0,6,0.008)
pz, pdz, coeffs_1z, coeffs_dz= tpoly(ztcp,ztcp+qf,t,0,0)
py, pdy, coeffs_1y, coeffs_dy= tpoly(ytcp,ytcp+qf,t,0,0)
px, pdx, coeffs_1x, coeffs_dx= tpoly(xtcp,xtcp+qf,t,0,0)


s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT_30003))

start = time.time()

while(flag):

    packet_1 = s.recv(4)
    packet_2 = s.recv(8)
    packet_3 = s.recv(48)
    packet_4 = s.recv(48)
    packet_5 = s.recv(48)
    packet_6 = s.recv(48)
    packet_7 = s.recv(48)
    packet_8_1 = s.recv(8)
    packet_8_2 = s.recv(8)
    packet_8_3 = s.recv(8)
    packet_8_4 = s.recv(8)
    packet_8_5 = s.recv(8)
    packet_8_6 = s.recv(8)
    packet_9 = s.recv(48)
    packet_10 = s.recv(48)
    packet_11 = s.recv(48)
    packet_12_1 = s.recv(8)
    packet_12_2 = s.recv(8)
    packet_12_3 = s.recv(8)
    packet_12_456 = s.recv(24)
    #packet_12 = s.recv(48)
    packet_13_1 = s.recv(8)
    packet_13_2 = s.recv(8)
    packet_13_3 = s.recv(8)
    packet_13_456 = s.recv(24)
    packets_extra = s.recv(600)

    # POSICIONES ARTICULARES

    q1 = str(packet_8_1)
    q1 = struct.unpack("!d", packet_8_1)[0]
    q2 = str(packet_8_2)
    q2 = struct.unpack("!d", packet_8_2)[0]
    q3 = str(packet_8_3)
```

```
q3 = struct.unpack("!d", packet_8_3)[0]
q4 = str(packet_8_4)
q4 = struct.unpack("!d", packet_8_4)[0]
q5 = str(packet_8_5)
q5 = struct.unpack("!d", packet_8_5)[0]
q6 = str(packet_8_6)
q6 = struct.unpack("!d", packet_8_6)[0]

# POSICION TCP

pos_x = str(packet_12_1)
pos_x = struct.unpack("!d", packet_12_1)[0]
pos_y = str(packet_12_2)
pos_y = struct.unpack("!d", packet_12_2)[0]
pos_z = str(packet_12_3)
pos_z = struct.unpack("!d", packet_12_3)[0]

# VELOCIDAD TCP

vel_x = str(packet_13_1)
vel_x = struct.unpack("!d", packet_13_1)[0]
vel_y = str(packet_13_2)
vel_y = struct.unpack("!d", packet_13_2)[0]
vel_z = str(packet_13_3)
vel_z = struct.unpack("!d", packet_13_3)[0]

q = [q1, q2, q3, q4, q5, q6]

end = time.time()
duration = end - start

if duration < 6:
    interval = end - start
    posiciony = ytcp
    velocidady = 0
    posicionx = xtcp
    velocidadx = 0
    posicionz = np.polyval(coeffs_1z, interval)
    velocidadz= np.polyval(coeffs_dz, interval)
    pdotZ=velocidadz+K*posicionz-pos_z
    pdotY=velocidady+K*posiciony-pos_y
    pdotX=velocidadx+K*posicionx-pos_x
    poszz.append(posicionz)
    velzz.append(velocidadz)
    posyy.append(posiciony)
    velyy.append(velocidady)
    posxx.append(posicionx)
```

```
        velxx.append(velocidadx)
        pos_Z.append(pos_z)
        vel_Z.append(vel_z)
        pos_Y.append(pos_y)
        vel_Y.append(vel_y)
        pos_X.append(pos_x)
        vel_X.append(vel_x)


    if duration > 6 and duration < 12:
        interval = end - (start+6)
        posicionz = ztcp+0.1
        velocidadz = 0
        posicionx = xtcp
        velocidadx = 0
        posiciony = np.polyval(coeffs_1y, interval)
        velocidady= np.polyval(coeffs_dy, interval)
        pdotZ=velocidadz+K*posicionz-pos_z
        pdotY=velocidady+K*posiciony-pos_y
        pdotX=velocidadx+K*posicionx-pos_x
        poszz.append(posicionz)
        velzz.append(velocidadz)
        posyy.append(posiciony)
        velyy.append(velocidady)
        posxx.append(posicionx)
        velxx.append(velocidadx)
        pos_Z.append(pos_z)
        vel_Z.append(vel_z)
        pos_Y.append(pos_y)
        vel_Y.append(vel_y)
        pos_X.append(pos_x)
        vel_X.append(vel_x)


    if duration > 12 and duration < 18:
        interval = end - (start+12)
        posicionz = ztcp+0.1
        velocidadz = 0
        posiciony = ytcp+0.1
        velocidady = 0
        posicionx = np.polyval(coeffs_1x, interval)
        velocidadx= np.polyval(coeffs_dx, interval)
        pdotZ=velocidadz+K*posicionz-pos_z
        pdotY=velocidady+K*posiciony-pos_y
        pdotX=velocidadx+K*posicionx-pos_x
        poszz.append(posicionz)
        velzz.append(velocidadz)
```

```
        posyy.append(posiciony)
        velyy.append(velocidady)
        posxx.append(posicionx)
        velxx.append(velocidadx)
        pos_Z.append(pos_z)
        vel_Z.append(vel_z)
        pos_Y.append(pos_y)
        vel_Y.append(vel_y)
        pos_X.append(pos_x)
        vel_X.append(vel_x)


    J = robot.jacob0(q)
    Jacobiana = np.around(J, 3)

    qdot = np.matmul(np.linalg.inv(Jacobiana),np.transpose([pdotX, pdotY, pdo
tZ, 0, 0, 0]))
    qd1 = qdot[0]
    qd2 = qdot[1]
    qd3 = qdot[2]
    qd4 = qdot[3]
    qd5 = qdot[4]
    qd6 = qdot[5]


    string = 'speedj([%f, %f, %f, %f, %f, %f], 20, 0.2)' % (qd1,qd2,qd3,qd4,q
d5,qd6) + '\n'
    strcode = string.encode()
    s.send (strcode)

    if duration > 18:
        flag  = 0

    i = i+1

save_duration = np.arange(0,18-(18/i),18/i)



plt.figure(1)
plt.plot(save_duration, poszz, 'tab:blue', label='Posicion Deseada')
plt.plot(save_duration, pos_Z,'tab:orange', label='Posicion Real')
plt.legend()
plt.xlabel('tiempo')
plt.ylabel('posicion en z')
```

```
plt.figure(2)
plt.plot(save_duration, velzz, 'tab:blue', label='Velocidad Deseada')
plt.plot(save_duration, vel_Z, 'tab:orange', label='Velocidad Real')
plt.legend()
plt.xlabel('tiempo')
plt.ylabel('velocidad en z')

plt.figure(3)
plt.plot(save_duration, posyy, 'tab:blue', label='Posicion Deseada')
plt.plot(save_duration, pos_Y,'tab:orange', label='Posicion Real')
plt.legend()
plt.xlabel('tiempo')
plt.ylabel('posicion en y')

plt.figure(4)
plt.plot(save_duration, velyy, 'tab:blue', label='Velocidad Deseada')
plt.plot(save_duration, vel_Y, 'tab:orange', label='Velocidad Real')
plt.legend()
plt.xlabel('tiempo')
plt.ylabel('velocidad en y')

plt.figure(5)
plt.plot(save_duration, posxx, 'tab:blue', label='Posicion Deseada')
plt.plot(save_duration, pos_X,'tab:orange', label='Posicion Real')
plt.legend()
plt.xlabel('tiempo')
plt.ylabel('posicion en x')

plt.figure(6)
plt.plot(save_duration, velxx, 'tab:blue', label='Velocidad Deseada')
plt.plot(save_duration, vel_X, 'tab:orange', label='Velocidad Real')
plt.legend()
plt.xlabel('tiempo')
plt.ylabel('velocidad en x')

ex = np.array(pos_X)-np.array(posxx)
ey = np.array(pos_Y)-np.array(posyy)
ez = np.array(pos_Z)-np.array(poszz)
exx = np.array(vel_X)-np.array(velxx)
eyy = np.array(vel_Y)-np.array(velyy)
ezz = np.array(vel_Z)-np.array(velzz)
```

```
plt.figure(7)
plt.plot(save_duration,ez , 'tab:orange', label='Error z')
plt.legend()
plt.xlabel('Tiempo (s)')
plt.ylabel('Error en Posicion (rad)')


plt.figure(8)
plt.plot(save_duration,ezz , 'tab:orange', label='Error z')
plt.legend()
plt.xlabel('Tiempo (s)')
plt.ylabel('Error en Velocidad (rad)')

plt.figure(9)
plt.plot(save_duration,ey , 'tab:orange', label='Error y')
plt.legend()
plt.xlabel('Tiempo (s)')
plt.ylabel('Error en Posicion (rad)')

plt.figure(10)
plt.plot(save_duration,eyy , 'tab:orange', label='Error y')
plt.legend()
plt.xlabel('Tiempo (s)')
plt.ylabel('Error en Velocidad (rad)')

plt.figure(11)
plt.plot(save_duration,ex , 'tab:orange', label='Error x')
plt.legend()
plt.xlabel('Tiempo (s)')
plt.ylabel('Error en Posicion (rad)')

plt.figure(12)
plt.plot(save_duration,exx , 'tab:orange', label='Error x')
plt.legend()
plt.xlabel('Tiempo (s)')
plt.ylabel('Error en Velocidad (rad)')

plt.show()
```